PARALLEL SIMULATION AND MULTIPLE-PATH EXECUTION TECHNIQUES FOR CHIP-MULTIPROCESSOR ARCHITECTURES

By

MATTHEW CHIDESTER

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2001

For Tiffany...

ACKNOWLEDGMENTS

I would like to acknowledge my committee chair, Dr. Alan George, for his guidance and support through my graduate education and for convincing me this whole thing would be worthwhile. I also would like to thank the other members of the High-performance Computing and Simulation (HCS) Research Lab for keeping things interesting. I especially thank Robert Todd and Matt Radlinski for their help in both technical matters and as a sounding board in difficult times.

Most of all, I wish to thank my wife, Tiffany, for spending countless late nights in the lab with me and making sure that I never gave up. To her, I owe much more than this dissertation, but it is a wonderful start.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

PARALLEL SIMULATION AND MULTIPLE-PATH EXECUTION TECHNIQUES FOR CHIP-MULTIPROCESSOR ARCHITECTURES

By

Matthew Chidester

August 2001

Chairman: Dr. Alan George
Major Department: Electrical and Computer Engineering

Integrating multiple processing elements onto a single integrated circuit to form a chip-multiprocessor (CMP) has been proposed as a solution to the problem of increased wiring delays between elements of a integrated circuit. This dissertation exploits the architecture of a CMP to both reduce the simulation time required to study such chips and increase the performance of applications running on such a device.

The complexity of parallel systems has increased both the need for comprehensive simulation and the computation time required to perform the simulations. CMP architectures are particularly susceptible to this effect, combining the requirements of a microprocessor simulator with that of a parallel system. In the first part of this dissertation, a portable, distributed simulator for CMPs is developed and presented based on the Message Passing Interface (MPI) that is designed to run on a cluster of workstations. Because the simulator itself is a complex application, microbenchmark-based evaluation is used to compare parallelization algorithms and interconnects for use

in the parallel simulator while identifying potential bottlenecks. The best combination is shown to yield speedups of up to 16 on a 9-node cluster of dual-CPU workstations.

The tight coupling of processing units in a CMP allows new forms of parallelism to be exploited. The second part of this dissertation studies multiple-path execution (MPE) on a CMP design to provide speedup on unmodified sequential code by exploring different paths of a conditional branch on separate processors. The impact on MPE performance due to processor complexity and count, cache and branch prediction architecture, processor-to-path allocation strategies, and limited interprocessor communication capabilities is explored. Simulation shows 12.7% speedup of instructions per cycle (IPC) on SPECint95 with up to 33.5% on benchmark components with poor branch prediction accuracy. This level of speedup is achievable on an 8-processor, 8-issue CMP with a simple mesh interconnect with realistic latencies and limited bandwidth.

CHAPTER 1
INTRODUCTION

Rapid advancements in integrated circuit technology over the past several decades have resulted in an exponential increase in the number of transistors available on a single chip. Using these transistors, computer architects have been able to create processors with innovative features to exploit instruction-level parallelism, increasing performance far beyond that of simple clock-frequency scaling. However, two key limitations threaten to flatten the performance growth curve to which processor architects have become accustomed: increased wiring delay and simulation explosion.

Increased wiring delay refers to the phenomenon that as devices scale downward in size, interconnect delay begins to dominate the critical-path timing which determines the maximum clock frequency. Whereas current processor designs are often monolithic in nature, with long signal paths that span much of the chip, future processor architectures will need to have good signal locality to operate at the optimal clock frequency [MAT97].

Simulation explosion affects the processor design cycle at many levels. Not only must processor designers consider an ever-growing number of architectural features and alternatives, but increasingly complex chip layouts must be rigorously validated prior to costly and time-consuming fabrication runs. For future processor architectures of increasing complexity to be designed and tested at present design-cycle rates, the speed of simulation must increase as well.

One architectural approach that addresses both of these issues is the Chip-Multiprocessor (CMP) [HAM97a]. A CMP contains multiple processors on a single die, where the majority of data flows locally within a single processor. Cross-chip or global signaling is used only for inter-processor communication. Therefore, a CMP shows signal locality that is conducive to future sub-0.1 μm designs. Furthermore, a CMP simulation lends itself well to parallelization. Increasingly complex designs with larger numbers of on-chip processors can be simulated in an approximately constant time by scaling the number of processors on the simulation platform.

Research into chip-multiprocessing is a relatively new area. Most previous work has focused on exploiting traditional parallelism at increasingly fine granularities. New forms of parallelism such as thread-level and loop-level parallelism can be exploited in a CMP because inter-processor communication (IPC) has a relatively low overhead. In order to exploit such parallelism in traditional sequential code, programmer or compiler interaction is required. Unmodified sequential code would only be able to use a single processor of the CMP. Since CMP designs may sacrifice processor complexity for a higher processor count, the performance impact on sequential code may be undesirable.

In the first phase of this dissertation, parallel simulation techniques for CMPs are examined. Sequential methods for simulation of parallel systems are combined with traditional, event-driven parallel simulation to produce a CMP simulator that reduces simulation time by distributing the simulation of each processor to separate physical nodes on a cluster of workstations. The design of the simulator is selected among several alternatives with the aid of microbenchmarks. The microbenchmarks approximate the behavior of a fully-implemented simulator with significantly reduced design complexity,

enabling the study of a wide range of design options including parallelization algorithms, synchronization approaches, and cluster interconnects.

In the second phase of this dissertation, a novel approach to executing unmodified sequential code on a CMP called multiple-path execution is explored. Multiple-path execution applies idle processors of a CMP-based system to explore both paths of conditional branches in an otherwise serial program. By effectively reducing the branch misprediction ratio, performance can be increased. A mechanism by which unused processors are synchronized to the state of an occupied CPU is developed to allow low-overhead path forking in support of support multiple-path execution.

Finally, an overview of the architectural requirements for supporting other types of parallelism and speculation on a CMP is provided. The hardware requirements of such systems are compared with that required to provide multiple-path execution to determine the feasibility of systems which can support both types of performance enhancements.

The remainder of this dissertation is organized as follows. In Chapter 2, the circuit- and simulation-level limitations to future processor architectures are described in detail and several alternative architectures are introduced and compared to a CMP. In Chapter 3, the parallel simulation of a CMP is examined. Chapter 4 presents a simulative study of the proposed multiple-path execution mechanism. Both of these chapters begins with a description of the problem and related research, describes the implementation and key issues, and summarizes the results obtained. Chapter 5 relates other CMP-based studies to the requirements of multiple-path execution. Chapter 6 concludes with a summary of the research.

CHAPTER 2
BACKGROUND

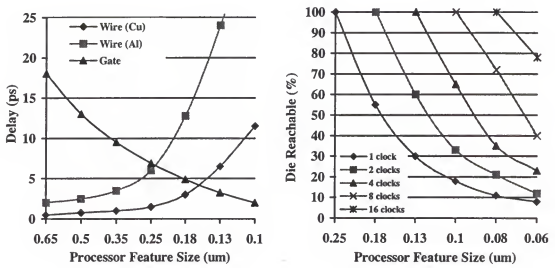
In this chapter, the circuit-level limitations on future, large-scale integrated circuits is examined in more detail to illustrate the dependence of future processor architectures on signal locality. The potential for simulation explosion is also described. Then, several alternative architectures are presented and compared with respect to their signal-local properties and simulative requirements. Finally, the architecture of a CMP is presented in greater detail.

2.1 Physical Limitations to Processor Architectures

As feature size is reduced in integrated circuits, transistors become smaller, faster, and draw less power. Therefore, more of them can be packed onto one die. Wiring also becomes narrower and often more metal layers are added to further reduce the chip area required to route signals. However, scaling has the opposite effect on the wiring as it does with transistors when it comes to switching speed. Thinner wires have higher resistance while adding metal layers and packing wires closer together increase the capacitance of the interconnects.

These effects contribute to a higher RC delay in the wiring. Figure 1a shows the interaction of transistor and wiring delays as integrated circuit technology scales to smaller sizes [KEC98]. The gate delay of the transistors steadily decreases while the wiring delay is exponentially increasing. New technology such as copper interconnects shifts the contribution of wiring delay, but the overall trend is the same. As the figure demonstrates, modern technologies of 0.18 or 0.13 μm yield aluminum wiring delays that

are greater than that of the transistors. Therefore, the clock speed of future microprocessors will be increasingly dependent on the wiring delay.



a) Sources of delay [KEC98]. b) Signal propagation [MAT97]

Figure 1. Physical limitations in future integrated circuits

Figure 1b shows the effect of increased wiring delay from a different perspective [MAT97]. The data in this figure assume that the maximum clock frequency is set to some fixed number of gate delays. As integrated circuits are scaled downward in size, this clock frequency increases as transistor switching times decrease. However, due to wiring delays, signals will be unable to cross the entire surface of the die in a single cycle. For example, at a feature size of 0.08 μm, only about 10% of the chip is reachable in one clock cycle. Conversely, 16 clock cycles are necessary for a signal to reach the entire chip. Possible solutions to this problem include lowering the clock frequency below its optimum value or limiting the die size. Such solutions are undesirable because they limit performance or functionality below the potential allowed by the process technology.

In order to accommodate cross-chip signaling while still maintaining optimal clock frequencies and die sizes, future microprocessor designs will have to be cognizant of signal locality. Several processor architectures have been proposed to take advantage of future billon-transistor integrated circuits with varying levels of signal locality. It is estimated that this level of integration will be possible at the 0.1 μm level [MAT97], where only 16% of the chip will be reachable in one clock cycle. Several processor architectures for billion-transistor designs will be presented later in this chapter and compared with respect to their signal locality.

2.2 Simulative Limitations to Processor Architectures

In addition to the circuit-level issues that will impact future processor architectures, designers are also facing the problem of simulation explosion. In the typical design cycle of a microprocessor, three levels of simulation are required: architectural, functional, and circuit-level.

At the architectural level, high-level simulations are performed to make basic decisions about cache size, issue width, pipeline depth, etc. and measure the performance tradeoffs of different architectural decisions. For each architectural design choice, many input programs must be simulated to determine the effect on a large variation of programs. As designs increase in architectural complexity, not only do the simulations require more computation time, but more simulations are necessary to investigate each architecture change.

The purpose of functional simulation is to verify the logic-level design of the processor using the parameters determined by the architectural simulation. At this stage, a large set of input programs must be tested to insure that all possible datapaths and instruction dependencies are handled correctly. Once again, as the architectural

complexity of the processor increases, the functional simulation requires both more processing time for each iteration and more iterations to test different types of input data.

Circuit-level simulation verifies the mapping of the logic-level design to physical transistors and wiring. Simulations are typically run using a SPICE-like simulator, first on basic schematics, then on schematics that are extracted from the VLSI layout of the chip. As in functional simulation, many types of input programs are necessary to verify all of the logic. Circuit-level simulations not only suffer from increased processing time due to the sheer increase in transistor count of successive processor generations, but also require more computational complexity per transistor. As transistors scale to smaller dimensions, first- or second-order approximations are replaced with more accurate mathematical and statistical expressions for the physical behavior of the devices.

In order to design future microprocessors, this simulation explosion will have to be controlled. Faster simulations will allow a designer to have more confidence in a design's correctness by testing more input data and will also permit the exploration of a larger number of design choices. Parallelization techniques [GEO96, MUK97] have been used to reduce the simulation time at the architectural level; similar techniques can be applied at the logical- and circuit-levels. Chapter 3 proposes a method by which the CMP architecture described for this dissertation can be parallelized for better simulation performance.

2.3 Processor Architecture Alternatives

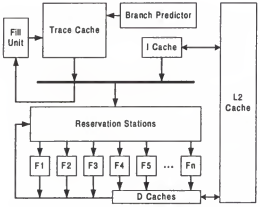
Figure 2 shows four alternative processor designs that make use of a billon-transistor chip design with varying degrees of signal locality and exploitable simulation parallelism. Each of the designs in this figure will be described in terms of its logical function and examined for signal locality and simulation parallelism.

Figure 2a shows how a modern superscalar processor could be extended to make use of a larger number of transistors [PAT97]. The design emphasizes large L2 and trace caches on the order of several megabytes each. The trace cache substitutes for a traditional instruction cache by using branch prediction logic to resolve conditional branches, storing instructions in program order rather than in memory order. To accommodate dozens or more functional units, thousands of reservation stations are also provided. The reservation stations act as an instruction "window." The issue logic can select from this window to find instructions that are independent of one another for simultaneous issue.

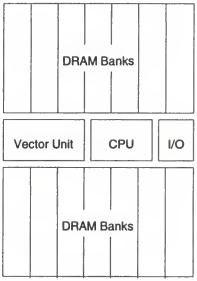
Such a superscalar design has several advantages. First, the architecture is well understood and is simply an extension of existing, proven technologies. Second, the performance of all programs can be increased without compiler or programmer intervention. However, this performance increase depends on how much instruction-level parallelism can be extracted from a single, sequential program. Studies have shown that such parallelism is limited [HEN96]. Furthermore, the complexity necessary to search thousands of instructions per clock cycle to find independent instructions for parallel execution is enormous.

Perhaps even more limiting is the fact that such a monolithic design has poor signal locality. The L2 cache, trace cache, branch prediction logic, and reservation stations all require significant amounts of chip area. These structures may limit the clock frequency to levels that are lower than the maximum allowed by the technology. Furthermore, simulators for superscalar processors are not typically parallelized because they are not easily broken into granularities that are coarse enough to show speedup on conventional parallel machines. Instead, designers employ techniques such as trace-

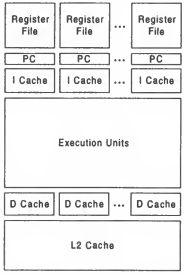
driven execution, instruction sampling, and careful optimization to provide fast simulation. While such techniques are effective at the architectural level, the logic- and circuit-level simulation of superscalar processors cannot take such shortcuts.



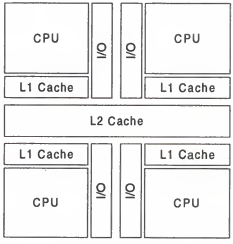
a) Future superscalar architecture



b) Processor with integrated DRAM



c) Multithreaded architecture



d) Chip multiprocessor

Figure 2. Processor architecture alternatives

Another potential processor architecture is that of integrated DRAM and processing logic or IRAM [KOZ97]. Figure 2b shows such a design. A billion-transistor die could accommodate a processor similar in complexity to a 4-way RISC microprocessor with 96 MB of integrated DRAM. The design could also include additional processors or special-purpose devices such as vector processing. By integrating the DRAM with the logic, the bandwidth and latency for memory access can be reduced. If additional DRAM is required, the on-chip DRAM can effectively act as an L2 or L3 cache for a larger set of off-chip memory.

The IRAM design has good signal locality for the processing unit(s). Processor complexity can be limited to a size that will allow the maximum clock frequency to be used. The DRAM portions would have poor signal locality, but tolerating DRAM access latencies is a common and well-studied requirement for microprocessors. IRAM would also have low simulation complexity. The relatively simple processor can be simulated in reasonable time using existing techniques. The structure of the DRAM comprising the majority of the chip area is very regular, allowing for both a short design time and piecewise simulation.

A potential drawback to IRAM devices is that the optimal fabrication process for DRAM devices is not the same process which produces the fastest logic devices, so some tradeoffs must be made. The largest issue with IRAM, however, is whether on-chip DRAM will provide a significant performance advantage over a conventional design with a large enough on-chip cache. SRAM cache access is significantly faster than a DRAM access, and with a large enough cache, the miss rate is on the order of a few percent. For this reason, most IRAM designs are targeting streaming media applications where a

combination of poor cache hit rates and vector processing would favor a design such as that in Figure 2b.

The multithreaded (MT) architecture in Figure 2c is another proposal for future processor designs [EGG97]. In an MT machine, different programs or multiple independent threads of a single program share a pool of functional units. The instruction fetch and decode mechanisms and the register files are duplicated for each running thread. Such a design simplifies the dependency checking requirements of the issue logic. A limited degree of instruction-level parallelism is extracted from each running thread, but instructions from multiple threads can be issued simultaneously to make efficient use of a large set of functional units. Supporting the execution of instructions from different threads at the same time is often referred to as simultaneous multithreading (SMT).

SMT architectures are able to provide performance gains in a system that often has multiple tasks running concurrently. Other mechanisms, such as the multiscalar architecture [SOH95], have been proposed which use compiler tools or machine-language translation to automatically parallelize sequential code into multiple threads that can run on an SMT architecture. SMT architectures also can be parallelized in simulation, though the shared functional unit pool represents a potential hotspot. The design of an MT processor has limited signal locality, however. While the fetch logic, register files, and L1 caches have good signal locality, the single shared pool of functional units may limit the clock frequency.

Figure 2d shows the architecture of a chip multiprocessor (CMP) [HAM97a]. A CMP features multiple processors on a single die. The processors share some level of cache but are otherwise independent. A CMP can execute conventional, explicitly-

parallel program code or it can exploit parallelism at a finer granularity. Such applications are explored in Chapters 4 and 5 of this dissertation.

A CMP has several advantages over the other designs in Figure 2. First, signal locality is excellent. The size and complexity of the processors can be selected to allow an optimal clock frequency. Simpler processor designs permit more processors to fit on a single die. The cache hierarchy can also be altered to accommodate signal locality by changing both the number of processors sharing a cache and the number of levels in the hierarchy.

The symmetric nature of a CMP is favorable towards simulation of such a device. Parallelization techniques such as those proposed in Chapter 3 can be used at the architectural level to validate the design. At the logic and circuit levels, the individual processors can be verified in high detail, and then reproduced in a similar manner as the cells of a DRAM design.

This dissertation will explore the design of a CMP with the goals of maintaining signal locality and providing scalable simulation support. Architectural features are added to a CMP to provide performance enhancements to sequential programs. The relationship of these enhancements to other work concerning parallel code executing on a CMP will be examined. The additional features will preserve the signal locality of the CMP design by assuming a limited bandwidth and non-zero latency for inter-processor communication.

CHAPTER 3
PARALLEL SIMULATION

Simulation is an important tool in the development of new microprocessors. A design must be rigorously validated prior to implementation to ensure that it is both functionally correct and performs well. A large range of test cases must be explored while many configurations are compared to optimize the design. As microprocessors become more complex, encompassing both a larger number of transistors and numerous architectural features for increased performance, the design space that must be evaluated through simulation explodes.

In the design of parallel systems, simulation times are increased further by the need to simulate multiple processors, a still wider range of inputs, and larger datasets. One technique for reducing the simulation time is to scale datasets down in size [CUL99], but this approach introduces inaccuracies and necessitates a detailed analysis of each workload to determine which part(s) can be safely scaled. Another performance enhancement involves simplified processor models with an emphasis on accurate memory subsystem and interconnect simulation [MUK97]. However, these simplified models do not reflect the behavior of modern superscalar, out-of-order processors. A third method to address simulation explosion is to make use of parallel simulation.

Parallel simulation employs multiple processing nodes to increase the simulation rate. A common approach is to perform separate simulations for different settings of parameters—i.e., cache sizes, workloads, datasets, etc.—simultaneously on different processors of a parallel system. While this approach can greatly increase the throughput

of the simulations, it does not reduce the latency required for a single simulation to finish. Often, a designer desires rapid feedback regarding a specific change in order to guide future decisions. For such situations, low-latency turnaround time is preferable to high throughput. Parallel simulation of event-driven models such as logic- and circuit-level evaluation has been successful in reducing these computationally intensive tasks. However, many of the architectural design choices for modern uniprocessors must be made early in the design cycle with the aid of a fast "performance model" that is often written in C or C++ [REI98]. Such a model is difficult to parallelize due to its irregularity and high communication-to-computation ratio.

Future systems are likely to be composed of multiple processors integrated on a single die known as chip-multiprocessors (CMPs) [COD01, HAM97a], combining the challenges of uniprocessor designs with the simulation complexity of a parallel system. The CMP has been introduced as a method to reduce the impact of on-chip interconnect delay on clock frequency [KEC98] and as a means to reduce design time by making use of repeated, regular structures. A CMP also lends itself to parallel simulation: the processors in a large CMP can be distributed to different nodes of a parallel machine and simulated simultaneously.

This chapter explores the parallel simulation of a CMP on a distributed system consisting of commercial off-the-shelf (COTS) workstations connected with a high-speed network. Such systems show a high level of cost-effectiveness when compared to a traditional, more tightly coupled parallel machine [AND95]. For portability, the simulator makes use of the Message Passing Interface (MPI) [MPI94] for inter-node communication. The performance of the parallel simulator on clustered workstations

connected by two popular system-area networks (SANs), the Scalable Coherent Interface (SCI) [SCI93] and Myrinet [BOD95], is explored.

In the next section, background information regarding conventional performance modeling of a CMP and common parallel simulation methods is introduced. Section 3.2 describes several parallelization approaches considered for the CMP simulation. In Section 3.3, the alternate approaches are studied through the use of MPI-based microbenchmarks running on the target platform to quantify the tradeoffs associated with each. Section 3.4 demonstrates the performance of a parallel CMP simulator based on the algorithm selected from the microbenchmarks. Section 3.5 provides a discussion of related research. Finally, Section 3.6 contains a summary of this work and avenues for future research.

3.1 Simulation of Multiprocessor Systems

In order to understand the design challenges of a parallel simulator for a CMP, it is important to consider the state-of-the-art in sequential simulations of uniprocessor and multiprocessor systems. Also, there are many approaches that can be used to parallelize a sequential simulation. In this section, both issues are addressed separately.

3.1.1 Sequential Simulation of a Microprocessor

Traditional, sequential simulation for performance modeling of a microprocessor typically achieves the fastest simulation rate by employing some form of trace-driven simulation. In trace-driven simulation, application code is run through a fast, functional simulator or on an existing machine (perhaps even of a different architecture from that being simulated). A "trace" of important activities is logged, consisting of a sequence of decoded instructions with specified input and output dependencies, memory accesses, and

branch instructions. The trace is then fed into a timing simulator to determine the execution time of the instruction sequence given stalls due to dependencies, cache misses, and branch mispredictions.

Because the trace files have a tendency to grow rather large, in practice they can be generated dynamically at the front-end of the simulator. This approach also allows simulation of mispredicted instructions, an effect that has been shown to cause cache pollution and significantly impact performance [SKA99]. Figure 3 shows a block diagram of a sequential simulator such as SimpleScalar [BUR97].

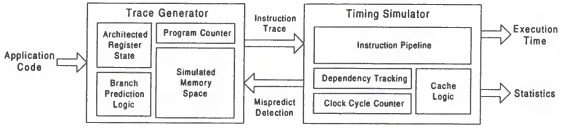


Figure 3. Trace-driven simulation of a uniprocessor using a sequential simulator

The front-end of the simulator maintains the programmer-visible register state including the program counter and the memory space occupied by the simulated application code. In order to simulate mispredicted instructions, branch prediction is also performed by the trace generator. In this respect, trace generation performs the same duties as the fetch stage in a pipelined microprocessor. The back-end consists of a timing simulator which feeds the trace through the complete pipeline, stalling on register dependencies and cache misses when appropriate. The timing simulator notifies the trace generator when a mispredicted instruction is retired so that a new trace can begin from the correct path.

The instruction trace is generated by executing each instruction as soon as it is fetched. By executing instructions in-order, the outcome of all branches can be immediately known. This technique allows simulation of key boundary conditions, such as perfect branch prediction. More importantly, it allows the remainder of the timing simulation to know a priori when an instruction is on a mispredicted path, greatly simplifying the recovery of speculative state when a mispredicted branch is resolved.

3.1.2 Sequential Simulation of a CMP

A chip-multiprocessor consists of multiple processors integrated on a single die. Unlike conventional symmetric multiprocessors (SMPs) that share only a common memory space, the processors in a CMP interface at a common cache level such as the L2 cache [OLU96]. An example of a 4-processor CMP is shown in Figure 4.

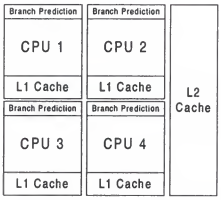


Figure 4. Organization of a CMP

To enable simulation of such a CMP, the simulation of each processor is interleaved on a clock-cycle basis. In a sequential simulation, the outermost loop simulates a single clock cycle for each CPU in the CMP before incrementing the clock-cycle counter and repeating the loop. This approach enforces time-ordered *consistency* between each processor in the CMP. By incrementing the cycle counter only when all

processors have completed execution of that cycle, any external effects which one processor may have on another are maintained.

In the CMP configuration shown in Figure 4, the only external effects which one processor may have on another involve accesses to the shared L2 cache and, by extension, the global memory. Maintaining *coherency* of each processor's view of this shared resource is another major issue in parallel simulation.

For the CMP architectures evaluated in this chapter, a Modified-Shared-Invalid (MSI) protocol [HEN96] is used to enforce coherency between the L1 caches and the shared L2 cache. A given cache line in the modified state indicates that only one L1 cache contains that data and both read and write access is permitted. A line in the shared state may be present in multiple L1 caches and is therefore read-only. If a processor desires write access to a shared line, it must first request exclusive access through the L2. The L2 cache will invalidate any other shared lines and grant the access. A line that is not present in any cache is said to be in the invalid state.

3.1.3 Parallel Simulation Techniques

Research in the area of parallel simulation techniques focuses primarily on event-driven simulation. In a parallel, event-driven simulation, each parallel process maintains a queue for locally generated events and separate queues for events generated by each remote process that will influence the local process. Events are processed in-order, with the event having the smallest timestamp of those in any queue, local or remote, processed first.

The challenge in parallel simulation is insuring that events are processed in globally consistent order. That is, an event can only be processed by the destination node when it can be sure that no events with an earlier timestamp will arrive at a later

time. One way to ensure such consistency is to wait until all remote input queues contain at least one event before selecting the one with the lowest timestamp.

Such a scheme can easily lead to deadlock situations when there are no events generated for a particular remote process. A common method to avoid this situation is to employ *lookahead* and *null messages* [CHA78]. If a process is deadlocked waiting for an event from one or more remote processes, it will send a null message to the other processes in the system. The null message represents an event that does not require any action but indicates the earliest timestamp for which an actual event may be received from the source node.

This timestamp is computed by taking the current time step of the source node plus a lookahead value. The lookahead value is system-dependent and represents any processing time that would be applied to incoming events before an outgoing event can be generated. An example of lookahead-based synchronization applied to a CMP will be provided in Section 3.2.

The use of lookahead and null messages can result in a significant level of generated null-message traffic, particularly if the lookahead is small or if the processes infrequently generate events destined for remote nodes [DEV90]. Another method is to use *barrier synchronization* [FUJ00]. In this approach, all processes process events freely up to a certain timestep and then wait for all other processes to reach the same timestep. If the barrier interval is selected based on the lookahead value, each process is guaranteed to have received any remote events that may influence the current barrier-bounded interval prior to the barrier at the beginning of the interval.

3.2 Parallel Simulation Approaches for a CMP

In this section, the parallel simulation techniques presented in the previous section are combined with simulation of multiprocessors to produce several alternatives for parallel simulation of a CMP. For performance reasons, architecture-level simulations of microprocessors are cycle-driven rather than event-driven. Typically, hundreds of events occur each clock cycle in a speculative processor, making the overhead of event queues too costly. Therefore, traditional event-driven parallel simulation is not directly applicable to a CMP.

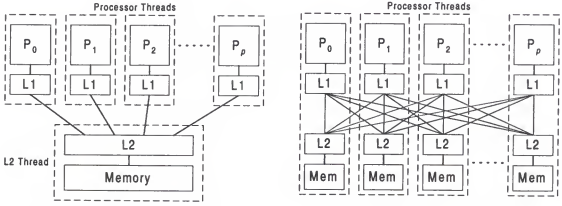
However, as noted previously, a CMP can be viewed as a collection of independent processors that can only affect one another through the memory hierarchy. If the architecture of Figure 4 is assumed, all such external events will occur through the shared L2 cache. While a single processor may access memory multiple times per clock cycle, the accesses frequently will hit in the L1 cache. With modern L1 caches providing hit rates in excess of 95%, accesses to the L2 cache are relatively infrequent.

In this dissertation, the proposed method for parallelizing CMP simulation is to apply conventional, cycle-driven simulation techniques to the processor and L1 caches, and parallel, event-driven simulation between each L1 cache and the shared L2. Furthermore, because L2 accesses are infrequent and have a high latency in terms of the simulated timespace, the simulation can be parallelized using a message-passing approach, allowing use of cost-effective and scalable distributed systems as the underlying simulation platform.

The following discussion assumes that the parallel simulation will be divided into one or more threads for concurrent execution. The mapping of threads to nodes on the simulation platform will be considered later.

3.2.1 Centralized vs. Distributed L2 Parallelization

One design choice for parallelization involves the shared L2 cache. The most straightforward approach is to simulate the L2 cache in a dedicated thread and each processor in a separate thread, as shown for a *p*-processor CMP in Figure 5a. The dotted lines in this figure indicate that each processor and associated L1 cache are simulated in a *processor thread* while the L2 and memory subsystems are handled by the *L2 thread*. It is possible for each processor thread to handle more than one processor/L1 pair when p exceeds the number of processor threads, N, although in this typical example, $N = p$ and the total number of threads is $p + 1$.



a) Centralized L2 simulation b) Distributed L2 simulation

Figure 5. Centralized vs. distributed simulation of the L2 cache

The centralized approach has several disadvantages. First, by requiring a thread dedicated to L2 transactions, the parallel efficiency is reduced. Parallel efficiency is defined as the speedup divided by the number of processors in the simulation platform, where speedup is the execution time of a sequential simulation divided by the execution time of the parallel simulation. Since the speedup is limited to N, the parallel efficiency is at most $N / (N+1)$ which is poor for small N. For large N, the centralized L2 becomes a

bottleneck. To alleviate the first problem, one of the processor threads can take on the dual role of simulating one or more processor/L1 pairs and the L2 cache. However, this approach limits the performance of the processor simulations of that thread, requiring the other processor threads to periodically wait on the slower thread.

An alternative approach is to distribute the simulation of the L2 cache across all processor threads and eliminate the L2 thread as in Figure 5b. While this figure assumes that $p = N$, each thread can simulate multiple processor/L1 pairs as well as part of the L2 cache when $p > N$. The memory space is interleaved on a cache-block basis with each thread having responsibility for all accesses to a particular bank of memory. If lower-order interleaving is used, the accesses should be relatively evenly distributed, allowing the performance to scale as N is increased.

One drawback to a distributed L2 simulation is the inability to simulate contention. For example, if the number of L2 cache ports must be limited to fewer than N, each portion of the L2 must communicate with the other portions to arbitrate for access. Another drawback involves maintaining a global view of the clock. With a centralized L2 simulation, all-to-one and one-to-all communication can be used to maintain clock synchronization, but a distributed L2 simulation requires costly all-to-all communication.

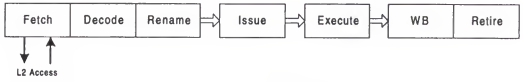
The performance of either approach depends on the frequency of cache accesses and clock synchronization events as well as the number of threads tasked to the simulation. These tradeoffs will be evaluated in greater detail in Section 3.3.

3.2.2 Blocking vs. Non-blocking L2 Requests

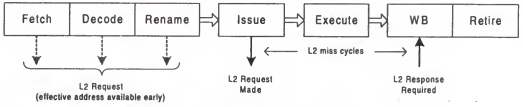
Another design decision with a large potential effect on parallel simulation performance involves the accesses to the L2 cache. Traditional, sequential simulators

applying a trace-driven approach as in Figure 3 decouple the memory access from cache hit checking. In simulations of CMP architectures, the cache access is integral to the data access due to the need to check coherency. Therefore, the trace generator must perform memory transactions through the cache logic.

Supporting trace-driven cache accesses requires the accesses to be blocking. That is, each memory reference is processed in order by the front-end of the simulator before moving on to the next instruction. Figure 6a demonstrates the placement of the L2 accesses of such a simulation on a processor with 7 pipeline stages similar to that which will be used in the simulations presented later in this chapter. Cache accesses are made during the fetch stage of each instruction and the processor thread blocks until a response from the remote L2 is received.



a) Blocking L2 Access



b) Non-blocking L2 Access

Figure 6. Blocking vs. non-blocking accesses to the L2 cache

As mentioned in Section 3.2, trace-driven simulation greatly simplifies the task of handling mispredicted instructions. However, requiring blocking access to the L2 cache in a parallel simulation is a significant disadvantage. The L2 transactions now initiate a

remote transaction and therefore have a much higher latency than in a sequential simulation. While the transaction is outstanding, no other instructions in the simulated processor can be fetched, limiting performance.

An alternative approach is to allow non-blocking accesses to the L2 cache as in Figure 6b. This approach more closely resembles the execution of instructions in hardware than the trace-driven approach. L2 requests can be made as soon as the effective address is available and an L1 cache miss is detected. The request can be performed as early as the fetch stage if the effective address is not dependent on any unfinished instructions. At the latest, the request is performed when all input dependencies are satisfied and the instruction is issued. Since an L1 miss will have a latency of several cycles in the simulated processor even if it hits in the L2, the simulation can continue for a number of cycles before the response from the L2 is required in the writeback stage. In this manner, the communication latency for the remote L2 transaction can be hidden.

The major disadvantage of a non-blocking approach is added complexity in the simulator. Trace-driven simulation can no longer be employed and branch mispredictions are not readily identifiable.

3.2.3 Lookahead vs. Barrier Synchronization

Both the lookahead and barrier synchronization approaches described in Section 3.2 can be straightforwardly applied to the interface between the L1 and L2 caches in a parallel CMP simulator. In both cases, the timestep for synchronization is based on the L2 cache access time in terms of simulated clock cycles.

For example, in a CMP system where events are generated between the processors and the shared L2 cache, the lookahead interval would be the L2 cache access time, L. If

the L2 has processed all events up to and including time t, the earliest time that an event destined from the L2 to a processor (e.g. a cache line invalidation) can be generated is $t + L$. The L2 could send a null message to each processor thread indicating that it is safe to proceed up to time $t + L$ without the possibility of receiving an event with an earlier timestamp from the L2.

Similarly, the barrier synchronization approach requires all threads to wait at a barrier every L clock cycles. In this manner, requests issued prior to a barrier can only affect events after the barrier. As long as messages are received in-order and the L2 processes all outstanding requests before reaching the barrier, consistency is preserved between all threads in the parallel simulation.

3.3 Evaluation of Simulator Design Alternatives

In order to select the best parallel simulator design from the alternatives presented in Section 3.2, each design decision must be studied with regard to performance tradeoffs. Developing a simulator for each alterative would be very costly in terms of design time and complexity. Instead, several MPI-based microbenchmarks were developed to enable study of the tradeoffs associated with the parallel algorithm alternatives when run on the target system. Key components of the full-fledged simulator are abstracted and provided as parameters to a much simpler parallel program that, in essence, simulates the behavior of the final application.

3.3.1 Evaluation Platform

The purpose of the microbenchmarks is to model the expected communication behavior of the parallelized simulator as closely as possible. In this section, we first consider the parameters of the simulator that are necessary to describe its communication

pattern. Since the purpose of the microbenchmarks is to study as many of the parallel algorithms as possible, we next introduce the design space that will be explored. Finally, we describe the experimental platform on which the microbenchmarks and, ultimately, the parallel CMP simulator will be executed.

Simulation parameters

As in most parallel applications, the performance of a parallel CMP simulation will be largely limited by the communication. As mentioned in Section 3.2, the communication will consist of two components: data-value communication between the separate L1 and shared L2 caches and clock-cycle synchronization. The latter is largely determined by the parallel algorithm and is a key component for microbenchmark evaluation. The former is a product of the simulated application and the timing simulator. For simplicity, the cache-to-cache communication pattern will be abstracted using parameters measured from a sequential version of the simulator.

The simplest model for the cache transactions involves only two parameters: the time required to simulate a single clock cycle for a single processor of the CMP, t_c, and the average number of cache transactions generated each cycle, A. The t_c parameter is defined in terms of the rate of a uniprocessor simulation in cycles/sec, f_c. The microbenchmark simulates the full application by simply delaying for t_c seconds per simulated clock cycle. Since experimental analysis shows A to be less than one, the microbenchmarks generate cache requests in a given clock cycle only when a randomly chosen value between zero and one is found to be less than A.

Additional parameters to the microbenchmarks include the number of processors in the simulated CMP, p, and the number of processor threads in the parallel simulation, N. The clock synchronization frequency depends on the lookahead, in this case the L2 cache

access latency, and is defined as L. The parameters and their default values, taken from a sequential version of the CMP simulator, are summarized in Table 1.

Table 1. Parameters to microbenchmark simulations

Parameter	Values	Description
t_c	$1 / f_c$ seconds	Simulation latency per clock cycle
f_c	100000 cycles/sec	Simulation rate of clock cycles
A	0.045 accesses/cycle	L2 cache accesses generated per cycle
L	12 cycles	Lookahead for parallel simulation
p	2 – 32 processors	Number of processors in simulated CMP
N	1 – 16 threads	Number of processor threads in parallel simulation

The t_c value is dependent on the performance of the sequential simulator. Smaller values of t_c indicate a faster sequential simulator speed and therefore a more difficult parallelization task due to a higher communication-to-computation ratio. The parameter A is comprised of four types of accesses: L1 instruction-cache misses, L1 data-cache misses, L1 data-cache writebacks, and L1 coherence misses.

Algorithm alternatives

Using the microbenchmark tests, each of the three major design options presented in Section 3.2 will be explored: centralized vs. distributed L2 cache simulation, blocking vs. non-blocking L2 cache accesses, and lookahead vs. barrier clock synchronization. Table 2 illustrates the naming convention used in the remainder of this section to distinguish between design alternatives.

Note that not all possible combinations are present in the table; when appropriate, the design space has been narrowed through the tradeoff analysis presented later in this section. Also, three incomplete designs are examined: CB, CN, and DN. These configurations do not conduct any form of clock synchronization and therefore could not

be used in the full parallel simulator. They are presented as a baseline to establish the impact of clock synchronization communication relative to the data communication.

Table 2. Parallelization alternatives

Name	L2 Simulation	L2 Accesses	Clock Synchronization
CB	Centralized	Blocking	-
CBL	Centralized	Blocking	Lookahead
CBB	Centralized	Blocking	Barrier
CN	Centralized	Non-blocking	-
CNL	Centralized	Non-blocking	Lookahead
CNB	Centralized	Non-blocking	Barrier
DN	Distributed	Non-blocking	-
DNB	Distributed	Non-blocking	Barrier

Experimental platform

In the parallel simulator, the vast majority of messages are either cache requests, cache responses, or clock synchronization messages. A cache request contains an address field, a source processor identifier, and a timestamp for a total payload of 32 bytes. Messages that are considered cache requests include line fill and upgrade requests from an L1 to the L2 cache or line flush and downgrade requests from the L2 to an L1 cache. Cache responses consist of the same identifying fields as the cache request plus the associated data. Cache lines are 32 bytes, giving cache response messages a total payload of 64 bytes. Cache responses include both line fills from the L2 to a requesting L1 and writebacks from an L1 to the L2. Clock synchronization messages for lookahead-based synchronization contain only a 16-byte timestamp, while barrier synchronization allows use of zero-byte messages. The type of message is identified through the MPI_TAG field and therefore does not add to the payload length.

The SCI testbed consists of 9 nodes connected in a 3×3 unidirectional torus. Each node contains two 733-MHz Pentium-III processors using a Serverworks LE chipset and

256 MB of PC133 SDRAM. The SCI adapters are from Dolphin and Scali [SCA00] and feature a link speed of 3.2 Gbps with a 32-bit, 33-MHz PCI interface. Version 2.1.2 of Scali's implementation of MPI, ScaMPI, provides the messaging layer.

The Myrinet testbed consists of 9 nodes connected through an M2L-SW16 16-port switch. Each node features dual 600-MHz Pentium-III processors and an i840 chipset with 256 MB of PC100 SDRAM. The Myrinet adapters have a link speed of 1.28 Gbps with a 64-bit, 66-MHz PCI interface. GMPI 1.2.3 is used for the messaging layer.

The disparity in processor clock frequency between the SCI and Myrinet testbeds is negated by the fact that the per-cycle computational delay, t_c, is measured in absolute seconds. The microbenchmarks increment the simulated clock cycle counter at this rate, measured by a cycle-accurate counter. Therefore, both platforms assume the same uniprocessor simulation rate. Any differences measured in performance will be due to the communication delay alone.

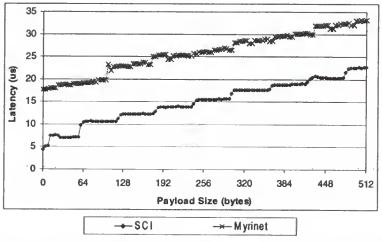


Figure 7. RTT/2 latencies for MPI messages over SCI and Myrinet

Figure 7 compares the latencies of SCI and Myrinet. The plotted data is for one-half of the round-trip time (RTT) of an MPI message of specified size. SCI provides a

much lower latency than Myrinet, particularly for small packet sizes. The 32-byte cache request messages have a latency of about 7.1 μs under SCI and almost 19 μs with Myrinet. The larger cache response messages require latencies of 10.7 μs for SCI and 19.1 μs for Myrinet. The small clock synchronization messages have an extremely low latency with SCI at 5.2 μs for an 8-byte message in the lookahead scheme or 4.5 μs for a zero-byte barrier message. Myrinet shows no advantage to very small messages, requiring about 17.6 μs for either size.

3.3.2 Microbenchmark Results

In this section, the microbenchmarks are used to determine the best design alternative from those discussed previously. First, the combinations of L2 simulation, access approach, and clock synchronization protocol will be examined through experiments run on the SCI testbed. Then, the performance of the optimal scheme will be compared against the same on the Myrinet testbed.

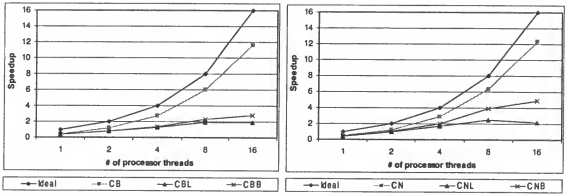
In the following experiments, it should be noted that N refers to the number of processor threads. For the distributed L2 simulations, there are N total threads in the parallel simulation. For the centralized L2 simulations, an additional thread is used for the L2 cache requiring $N + 1$ total threads. When parallel efficiencies are provided, the efficiency is in reference to the total number of threads as appropriate for the L2 simulation scheme.

The mapping of threads to nodes on the testbed is always one-to-one. The only exception is made when $N = 16$. Because the SCI testbed is limited to 9 nodes, two processor threads are run on each SMP node, sharing a single SCI interface. When a

centralized L2 is required, the L2 thread is run alone on the ninth node. The same configuration is used on the Myrinet testbed.

Parallelization schemes

The first set of microbenchmark experiments makes use of a centralized L2 cache. Figure 8a shows the performance with blocking L2 accesses while Figure 8b demonstrates non-blocking performance. Both figures compare the speedups obtained with lookahead and barrier synchronization. Two baselines are provided: an ideal speedup and the speedup when no clock synchronization is performed. The ideal speedup is defined as the number of processor threads.



a) blocking L2 accesses b) non-blocking L2 accesses

Figure 8. Microbenchmark performance predictions for centralized L2 cache schemes on SCI testbed

The first trend evidenced in Figure 8 is that the CB and CN approaches both track well with the ideal. Furthermore, there is very little difference between CB and CN because the L2 requests from each processor are very infrequent. However, a large reduction in performance results when clock synchronization is added. This reduction is due to the fact that processors must wait even when they do not have an L2 request of

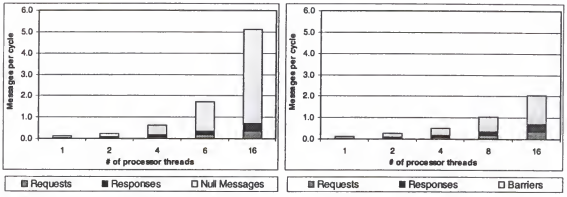
their own when other processors have requests. As p increases, the probability of at least one processor performing a request and therefore requiring the others to wait is increased.

With lookahead synchronization taken into account, Figure 8 shows a maximum speedup of about 2 for both CBL and CNL schemes with $N = 16$. Barrier synchronization fares much better, particularly with non-blocking accesses, showing a speedup of 3 for CBB and 5 for CNB. The benefit of non-blocking accesses in the barrier approach is that all processors can proceed to the barrier even if there are outstanding requests. With blocking accesses, any processor making an L2 request will arrive at the barrier much later than those that do not make a cache access. Once again, as p is increased, the probability of a single processor slowing the rest increases.

The difference between the performance with and without clock synchronization and the difference between the two types of synchronization can be explained by examining the type and number of messages that the L2 thread must handle. Figure 9 shows the number of messages sent and received by the L2 thread per simulated clock cycle. Figure 9a shows the number of cache requests, cache responses, and null messages processed in the CNL scheme, while Figure 9b shows the requests, responses, and barrier messages for CNB. In the simplified microbenchmark approach, the number of messages sent by the L2 equals the number received for each type. Also, the messages are evenly distributed between each of the processor threads.

As Figure 9 illustrates, the amount of traffic for clock synchronization greatly exceeds the cache traffic, particularly for large N. As indicated by Figure 7, SCI provides a latency of 10.7 μs for the large cache responses and lower latencies for the other messages. If all messages were of the larger variety, up to 93000 messages could be

serviced per second or 0.93 messages per simulated clock cycle. Thus, the performance becomes severely limited at $N \geq 8$ where more messages per cycle are required.



a) CNL b) CNB

Figure 9. Messages handled per simulated clock cycle by centralized L2 cache with non-blocking accesses on SCI testbed

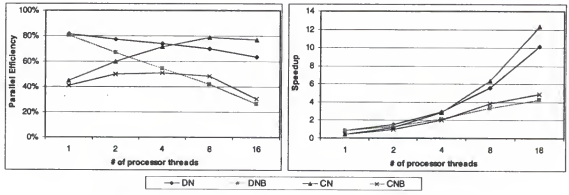
Figure 9 also illustrates a large difference between the number of null messages generated in CNL verses the number of barriers in CNB. The number of barriers scales linearly with N and is also dependent on L. Since the cache traffic also scales with N, the barrier messages account for a constant 65% of the total traffic for all values of N for the combination of L and A used.

By contrast, null messages are sent whenever a thread reaches a limit based on the lookahead. To understand why the number of null messages grows faster than N, consider a simple system with two processor threads, both initially at simulated clock cycle t_o. The first thread does not make any cache accesses and proceeds to cycle $t_o + L$ before sending a null message. The second thread makes a cache request in cycle $t_o + 1$. The L2 must wait until the first thread sends its null message before processing the second thread's request. At that time, it can respond to the first thread's null message with permission to proceed to cycle $t_o + 1 + L$. Therefore, the null message grants the

first thread permission to simulate only one additional clock cycle before another null message will be required.

As the number of threads in the system is increased, the probability of at least one thread making a request in any given clock cycle increases. Therefore, due to the effect described above, the null messages in CNL effectively grant permission for fewer clock cycles of simulation for increased *N*. Since the number of threads sending null messages increases and the null messages become more frequent, the number of null messages increases faster than *N*. For the data in Figure 9, null messages account for only 58% of the total traffic in a 2-thread system. In the 16-thread system, the null messages account for 86% of the total traffic.

The results from Figure 8 indicate that the combination of non-blocking L2 accesses and barrier synchronization perform best. In Figure 10, the distributed L2 approach is compared to the centralized L2 by using DNB and CNB schemes, respectively. For a baseline comparison, DN and CN performance is also shown. Figure 10a illustrates the parallel efficiency while Figure 10b shows the speedup.



a) parallel efficiency b) speedup

Figure 10. Microbenchmark performance predictions for centralized vs. distributed L2 with non-blocking accesses and barrier synchronization on SCI testbed

The efficiency of the distributed approach declines as the number of threads is increased, even for the DN scheme that does not perform clock synchronization. This decline in efficiency is due to the fact that the L2 processing task is distributed between all of the processor threads, thereby slowing the processor simulation. The CN scheme, by contrast, increases in efficiency due to the $N / (N + 1)$ scaling effect described earlier.

When barrier synchronization is added, the efficiency of the distributed approach decreases rapidly. This effect is due to the need to communicate a barrier message with every other thread in the system in DNB rather than with only the centralized L2 thread in CNB. This all-to-all synchronization requires N times as many transactions as the centralized approach.

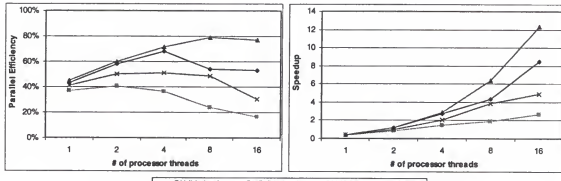
The resulting conclusion is that the distributed L2 has a slight performance advantage for a small number of threads, $N \leq 4$. At larger thread counts, the centralized L2 has a slight performance advantage. This outcome, coupled with the ability to efficiently model contention effects, gives the edge to a centralized L2 design.

Interconnect selection

With the conclusion that the CNB approach shows the greatest performance potential, a comparison of the underlying network for the parallel simulation platform is next performed. In Figure 11, the parallel efficiency and speedup of the CNB-based microbenchmarks are shown for both the SCI and Myrinet testbeds. The CN results are also provided as a baseline.

The performance of both interconnects with the CN baseline is similar up until $N = 8$, where the Myrinet platform shows a decrease in parallel efficiency. From Figure 9, it can be seen that at this system size, the L2 must process 0.36 request and response

transactions per simulated clock cycle. As previously discussed, the 10.7 µs latency of SCI allows up to 0.93 transactions per simulated clock cycle. Myrinet, however, shows a latency of 19.1 µs, allowing only 0.52 transactions per simulated cycle. Though this appears to be sufficient capacity for the required cache traffic, in practice the effective latency is increased due to contention when different processor threads make multiple requests simultaneously. At 0.36 transactions per clock cycle, the load on the Myrinet network exceeds 50% and contention is frequent. The SCI performance trails off similarly at $N = 16$ where 0.72 cache transactions are made per simulated clock cycle.



a) parallel efficiency b) speedup

Figure 11. Microbenchmark performance predictions for SCI vs. Myrinet testbed with centralized L2 and non-blocking accesses

When barrier synchronization is added, the performance advantage of SCI becomes even more pronounced. Not only can the SCI network handle a higher load due to its decreased latency, but the majority of the traffic consists of zero-byte barrier messages. As shown in Figure 7, ScaMPI provides a 4.5 µs latency for these messages while Myrinet requires 17.6 µs. The result is a potential speedup at $N = 16$ of 4.9 using SCI and only 2.7 with Myrinet.

3.4 Parallel Simulator Results and Analysis

To test the parallelization schemes described in the previous section, the SimpleScalar [BUR97] simulator was extended to produce *SimpleCMP*. SimpleScalar is a sequential, trace-driven simulator for uniprocessor architectures. It was first extended to simulate basic CMP architectures similar to Figure 4 with support for a CMP of up to 32-processors. Next, SimpleCMP was parallelized using the CNB scheme and subsequently executed on the SCI testbed. In this section, the simulation platform is described in greater detail and the parallel performance is explored. The measured performance is compared to the microbenchmark predictions from the previous section to demonstrate the validity of the microbenchmark approach.

3.4.1 Simulation Platform

SimpleCMP performs trace-driven simulations of a 32-bit, MIPS-like instruction set [PRI95], complete with register renaming, out-of-order execution, superscalar issue, and detailed cache and branch prediction logic. In addition to the CNB parallelization scheme, the simulator was also modified to more closely resemble the hardware of an Alpha 21264 [KES98] by replacing the register update unit (RUU) with a reorder buffer and rename logic. Separate issue queues are provided for integer and floating-point instructions, and the pipeline depth matches the design of the 21264. The dual, synchronized register-file of the 21264 was not implemented. Table 3 shows the key architectural parameters for each individual processor in the simulations.

In order to evaluate the parallel simulator under a variety of workload conditions, a representative subset of the SPLASH-2 benchmarks [WOO95] was selected. SPLASH-2 provides a suite of shared-memory benchmarks for parallel systems and includes several kernel and application components.

Table 4 lists the four benchmarks used, two kernels and two applications: *LU*, *ocean*, *radix*, and *raytrace*. The table also provides the measured values for f_c and A when executed on an 8-processor CMP using the sequential version of SimpleCMP. These values vary slightly for different degrees of parallelism, with f_c generally decreasing and A increasing with higher processor counts.

Table 3. Processor architecture parameters for parallel CMP simulation

Parameter	Value
Fetch/Issue/Retire Width	4 instructions
Rename Registers	128
Integer Issue Queue	64 instructions
Load/Store Issue Queue	32 instructions
Integer ALUs	4
L1 cache ports	2
Branch Predictions/cycle	1
Branch Predictor	*Global*: 12-bit history, 4k × 2-bit saturating counters *Local*: 1k × 10-bit history, 1k × 2-bit saturating counters *Select*: 12-bit global history, 4k × 2-bit saturating counters
Branch Target Buffer	2048 entries, 2-way associative
Return Address Stack	32 entries
Minimum Misprediction Latency	8 cycles
L1 I-Cache	64 kB, 2-way associative, 32-byte lines, 1-cycle latency
L1 D-Cache	64 kB, 2-way associative, 32-byte lines, 1-cycle latency
Unified L2 Cache	8 MB, 4-way associative, 32-byte lines, 12-cycle latency
Memory Access Latency	100 cycles, 8 bytes/cycle

All statistics are gathered only from the parallel portion of the benchmarks, neglecting any sequential start-up code and generation of working-set data. This common practice assumes that, with realistic workloads, the initialization code is insignificant compared to the overall execution time. In practice, the non-parallel part

can be sped up considerably by using only the front-end of the trace generator and not the timing simulator, or even eliminated entirely through checkpointing. Table 4 provides the approximate number of instructions that comprised the parallel portion of each benchmark.

Table 4. Selected SPLASH-2 benchmark components

Benchmark	Input Dataset	Instructions	8-processor Parameters	
			f_c	A
LU	256x256 matrix, 16k blocks	66 M	131k	0.0147
Ocean	258x258 grids, 4 time-steps	330 M	102k	0.0463
Radix	256k integers, radix = 1024	16 M	131k	0.0355
Raytrace	teapot.env	282 M	98k	0.0103
Average	-	174 M	107k	0.0267

As a baseline for comparison, Figure 12 shows the execution time for the sequential version of SimpleCMP for each benchmark and the overall average measured on one of the SCI testbed nodes. The execution time remains relatively constant up to an 8-processor CMP, then increases rapidly for 16- and 32-processor systems. This effect is due to the fact that the simulator does not fit well in the 256 kB L2 cache of the testbed systems for large processor counts.

The sequential simulation scheme described in Section 3.1 results in highly reduced locality at large processor counts due to the sequential iteration over each processor for every simulated clock cycle. In addition, the capacity required to hold all frequently used data increases. The key data elements for each iteration are the decoded instructions which are 236 bytes in length. Each processor can have up to 128 such

instructions in-flight, for a total of 29.5 kB of data. With eight processors in the simulated CMP, 236 kB is required to hold just the decoded instructions. In addition, the L1 caches and branch predictors are also accessed, but on a less frequent basis.

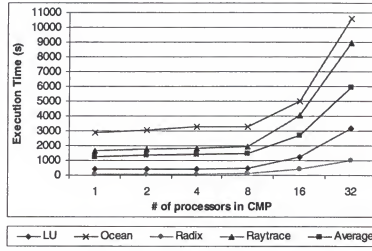


Figure 12. Sequential execution time for SPLASH-2 benchmarks

A 16-processor CMP simulation has twice as large a working set as an 8-processor CMP and therefore much poorer data locality. It is important to note that even the 32-processor CMP simulation fits in the 256 MB main memory of one of the testbed nodes, so swapping to disk is not an issue. Such a performance limitation would be relatively easy to fix by increasing the memory capacity, but cache sizes are much more difficult and expensive to increase.

3.4.2 Parallel Performance

The SPLASH-2 components from Table 4 were simulated on the parallel CMP simulator for a variety of CMP sizes and processor thread counts. Figure 13 shows the speedup obtained over the sequential version of the simulator. As before, the horizontal axis refers to the number of processor threads and does not include the L2 thread.

Several trends are apparent from the results in Figure 13. First, for CMPs of 8 processors and less, the speedup is relatively modest, reaching almost 4 for 8 threads. There is a slightly increased speedup for an 8-processor CMP over a 4-processor CMP even when both systems are simulated with the same number of threads. This increase is due to the fact that the synchronization overhead is a lower percentage of the total communication time when each thread simulates more than one processor. In effect, twice the number of instructions and twice the number of L2 cache requests can be simulated per barrier synchronization.

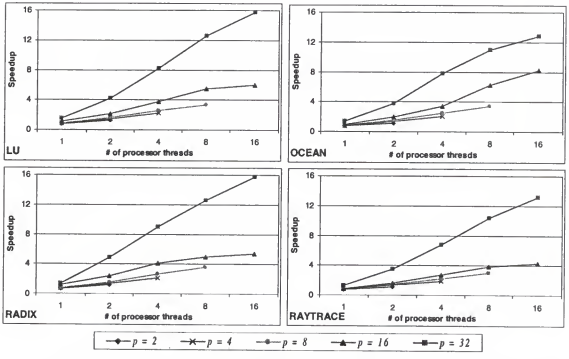


Figure 13. Speedup for selected components of SPLASH-2 for CMPs of varying size

For 16-processor CMPs, the speedup shows a somewhat larger improvement. At 32-processors, the performance is markedly improved. This effect can be explained by the improved cache locality in the parallel simulation. As illustrated in Figure 12, cache locality becomes problematic in a uniprocessor simulation of CMPs of this size. In the

parallel approach, however, a 16-thread simulation requires only one or two processors per thread for a 16- or 32-processor CMP, respectively. The parallelization effectively scales the size of the L2 cache with the number of threads, improving the performance beyond the simple computational overlap.

Figure 14 better illustrates this effect by showing the parallel efficiency (relative to the total number of threads, including the L2 thread). For system sizes of between 2 and 8 threads, parallel simulation of the 32-processor CMP shows a superlinear speedup due to the increased cache hit rate. At 16 threads, the reduced efficiency predicted by the microbenchmarks and illustrated in Figure 10 reduces the performance to sublinear levels.

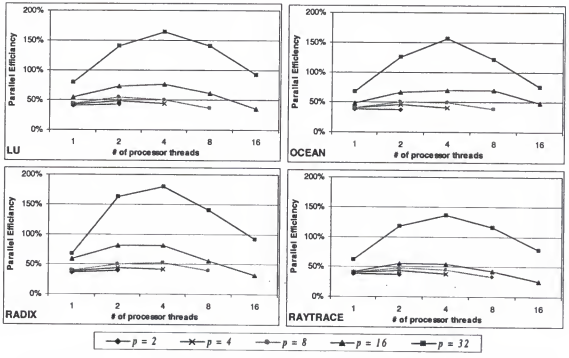


Figure 14. Parallel efficiency for selected components of SPLASH-2 for CMPs of varying size

Interestingly, the improved cache locality shows benefit even in the simplest case of a single processor thread plus the centralized cache node. The theoretical maximum efficiency is 0.5 in such a system, but the 32-processor CMP simulation exceeds this efficiency for all benchmarks while the 16-processor CMP simulation does so for the *LU* and *radix* benchmarks.

3.4.3 Comparison to Microbenchmarks

In order to assess the accuracy of the microbenchmarks in predicting the parallel simulator performance, microbenchmarks were run for each f_c and A value in Table 4 corresponding to one of the SPLASH-2 benchmarks and the results averaged. Due to the cache effects with CMP configurations of greater than 8 threads for which the microbenchmarks do not account, comparisons are made for an 8-processor CMP with thread counts from $N = 1$ to $N = 8$. The average result for the CNB-based microbenchmarks and the measured performance on the fully implemented simulator are shown in Figure 15.

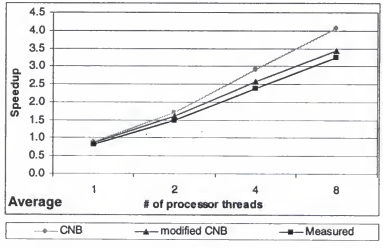


Figure 15. Comparison of microbenchmark prediction and actual parallel simulator performance

The CNB microbenchmark does not track particularly well with the actual parallel simulator performance. At $N = 1$, the difference is only 9.8%, but the disparity increases to 25% for $N = 8$. The disparity was found to be largely due to a neglected effect that has a non-negligible impact on the overall performance. Namely, cache requests from the L1 caches to the L2 cache can cause the L2 to require a writeback of dirty data from a second L1 in order to satisfy the first request. In the implementation used in the parallel simulator, the L2 blocks on such a request until the dirty line is returned by the second L1. Though the writeback transaction is included in the A parameter, the L2 cache blocking effect was not accounted for in the CNB microbenchmark.

The CNB microbenchmark was modified to accept another parameter: the probability that an L1 request will require a writeback request and concomitantly block in the L2. For the benchmarks studied, this probability ranged from 0.3% to 14% with an average of 9.3%. Using the measured probabilities for each SPLASH-2 component, the modified microbenchmarks were executed and the results plotted in Figure 15.

As illustrated in the figure, the modified CNB microbenchmark predicts the performance of the parallel simulator with much greater accuracy: within 10% of the measured results for the system sizes shown. The remaining difference can be attributed to the fact that the actual execution of the benchmarks frequently results in bursty L2 accesses, often synchronized with barriers or locks in the parallel code, reducing the performance as contention increases the latency for each transaction. By contrast, the microbenchmarks yield a more uniform distribution of cache accesses.

3.5 Related Research

Much existing research exists in the field of parallel simulation of multiprocessors. The Wisconsin Wind Tunnel (WWT2) project simulates a parallel, CC-NUMA system on various parallel systems including a Myrinet-connected cluster of workstations [MUK97]. Synchronized Active Messages provide the messaging layer rather than MPI. The WWT2 makes use of direct-execution simulation with each parallel process running on a separate CPU, capturing only the memory accesses and sending them through a sequential timing simulation of the memory hierarchy. This approach does not offer flexibility in the processor model and is unable to model effects such as different issue widths, speculative memory accesses, and out-of-order execution.

The Integrated Simulation Environment (ISE) takes a similar approach for simulation of MPI-based parallel programs [GEO98]. In ISE, MPI code executes natively on distributed nodes in a cluster with the MPI function calls being intercepted and sent to a centralized, sequential simulation of the interconnect. This "software-in-the-loop" approach allows flexibility in selecting the interconnect type and topology but is limited in performance by the relatively slow interconnect simulation.

In [GEO96], parallel simulation is applied to a parallel digital signal processor (DSP) system. In this study, the parallel programming language Linda is applied in a clustered environment to achieve speedup on a multi-DSP simulation despite the use of low-performance, 10 Mbps Ethernet as the interconnect.

The RSIM [PAI97] simulation environment provides great flexibility in the configuration of the individual processors in a simulated multiprocessor, but the simulation itself runs only on a uniprocessor. Rather than parallelizing the simulator, DirectRSIM [DUR99] provides a performance improvement by adapting RSIM to be

trace-driven instead of execution-driven. Using an approach similar to RSIM, the MINT [VEE94] multiprocessor simulator has even been modified to simulate a CMP [KRI98], but neither approach involves parallel simulation.

3.6 Summary

Simulation of parallel systems is an important tool in evaluating design alternatives. As the complexity of such systems increases down to the chip level with the advent of CMP-based systems, the simulation overhead becomes increasingly prohibitive. In this chapter, an MPI-based parallel simulation environment was presented to reduce the execution time of performance-level CMP simulations running on a cluster of workstations connected by a high-speed interconnect.

The parallel simulation is a natural extension of conventional, sequential simulation of a uniprocessor combined with traditional, event-driven parallel simulation techniques. The proposed shared-L2 design of future CMP architectures enables a parallelization approach with a relatively low communication-to-computation ratio. The CMP processor and L1 pairs can be locally simulated with an inherent reduction in the number of accesses that must be made to a remote L2 cache.

Several design alternatives were considered, including centralized vs. distributed simulation of the L2 cache, blocking vs. non-blocking L2 accesses, and lookahead vs. barrier clock synchronization. Through MPI-based microbenchmarks, the optimal combination was found to be a CNB scheme combining centralized L2, non-blocking accesses, and barrier synchronization with a predicted speedup of around 5 for 16 threads. The microbenchmarks also demonstrate that the low-latency communication

afforded by SCI makes it a more suitable interconnect for the parallel simulation platform than Myrinet.

Performance results for a fully implemented parallel simulator were presented for a variety of workloads. Due to increased cache capacity of the parallel platform, the results show higher-than-predicted speedups of between 12 and 16 when simulating a large CMP architecture with 16 processor threads on 9 dual-CPU cluster nodes. The parallel simulation results are shown to differ somewhat from the microbenchmark predictions even without the cache effect, but the difference is largely accounted for with a slight modification to the microbenchmarks.

Future work in this area could pursue several avenues. Larger system sizes, in terms of the numbers of simulated CMP processors, processor threads, and cluster nodes in the parallel simulation, could be studied. CMP processor counts were restricted to 32 in this study due to the fact that limitations in VLSI technology will likely yield CMPs of this size or smaller in the near future, but further advances or simpler processor designs may allow chip-level architectures with hundreds of processors. The cluster node counts and, by extension, processor threads in this study were limited by the hardware available.

Further study could also be directed at more efficient parallel simulation algorithms. For example, a more aggressive clock synchronization scheme might speculatively execute instructions past the lookahead or barrier clock cycle, rolling back the computation if an earlier-stamped message is received. The speculative nature of the processor pipeline being simulated would facilitate such an approach with little extra overhead.

Parallel simulation should increase in importance as parallel systems become larger and more complex. Fortunately, the existence of such systems will also enable parallel simulators to study the next-generation design alternatives with greater speed and efficiency. As the simulation application itself becomes more complex, techniques such as microbenchmark-based evaluation of design alternatives will become increasingly valuable in assessing the most effective parallelization technique for a given system.

CHAPTER 4
MULTIPLE-PATH EXECUTION

Rapid advancements in integrated circuit technology over the past several decades have exponentially increased the number of transistors available on a single chip. Using these transistors, computer architects create processors with innovative features to exploit instruction-level parallelism (ILP), improving performance beyond clock-frequency scaling. However, as mentioned in the previous chapter, increasing wiring delay at the silicon level and growing design and validation complexity threatens to quell this performance growth. The integration of multiple processors on a single die, known as a chip multiprocessor (CMP), has been proposed as a method to address both issues [COD01, HAM97a, KEC98, KRI98, OPL97].

A CMP provides the opportunity to exploit parallelism in programs beyond the ILP typically uncovered in a superscalar design and at granularities smaller than those achieved in a symmetric multiprocessor (SMP). One of the chief limitations to ILP in a traditional superscalar is branch prediction accuracy. As processor pipelines become longer and issue widths increase, the penalty associated with a mispredicted branch increases. In a CMP design, otherwise idle CPUs can be used to explore both paths of a conditional branch, ensuring that when the branch condition is eventually resolved, at least one CPU will have taken the correct path. This scheme is called multiple-path execution (MPE). MPE has been widely studied as a method to improve sequential program performance in superscalar and simultaneous multithreaded (SMT) designs [AHU98, KLA98, KLA99, UHT95, WAL98].

In this chapter, an architecture for providing MPE on a CMP is proposed. Other MPE studies have demonstrated speedups of 14% on SPECint95 with speedups in excess of 30% on components with poor branch prediction accuracy [AHU98]. These studies conclude that MPE is feasible when the number of explored paths is limited through application of confidence prediction [JAC96] to find branches that are likely to be mispredicted. Other pitfalls, such as the requirement for per-path return address stacks [AHU98], are also well documented.

Other architectural requirements for MPE unique to CMP designs will be presented. The key limitation is found to be the on-chip interconnect. Using a simulative analysis of an idealized CMP model, these key architectural elements are explored and the bandwidth and latency requirements demonstrated. Several options are proposed to reduce the interconnect dependency. Applying these techniques, practical, though forward-looking, CMP designs can yield speedups similar to those found in other MPE studies.

The remainder of this is structured as follows. In Section 4.1, the proposed architecture for enabling MPE on a CMP is described. Section 4.2 describes the simulation environment that is used to evaluate different MPE approaches. Specific architectural tradeoffs are explored and evaluated in Section 4.3, while the interconnect requirements are examined in Section 4.4. Practical limits to CMP designs are described in Section 4.5 and the MPE performance on several CMP implementations is compared. In Section 4.6, related efforts are described while Section 4.7 offers a summary of the work and opportunities for future research.

4.1 Multiple-path Execution on a CMP

Traditional MPE schemes rely on idle functional units and enhanced fetch and issue techniques to execute alternate paths. A CMP can achieve similar parallelism by assigning a different path to each idle processor. Confidence prediction determines if the predicted branch direction is likely to be correct. Provided there are sufficient resources, a low-confidence branch causes a second processor to begin executing down the not-predicted direction such that, regardless of the branch outcome, one processor follows the valid path.

It is highly likely that a system running multiple independent tasks or programs that have been parallelized in a more conventional manner will achieve greater system-wide speedup if processors are allocated to these tasks instead of to MPE. It will be assumed that there is some subset of the CMP that is idle and has been allocated to explore multiple paths of one of the tasks.

In the proposed scheme, a *primary* processor executes the code exactly as a uniprocessor would, following only the predicted path. At the start of execution, all other processors on the chip are designated as *free*.

When the primary processor encounters a low-confidence branch, a single free processor is assigned to follow the non-predicted direction and becomes an *alternate* processor. Though the alternate processors fetch and execute instructions, program order is preserved by permitting only the primary processor to commit its instructions. The committed register writeback operations are then broadcast to all other processors on the chip. In this manner, alternate processors receive data that may be input dependencies in their instruction stream while free processors maintain the state necessary to become alternates.

Figure 16 shows the proposed microarchitecture for an MPE-enabled processor on a CMP. An aggressive out-of-order design with in-order front and back ends, adapted from [KLA99] and similar to modern speculative designs, is assumed. MPE support requires four types of data to be communicated between processors: *MPE fork*, *dependency sync*, *value sync*, and *MPE resolve*.

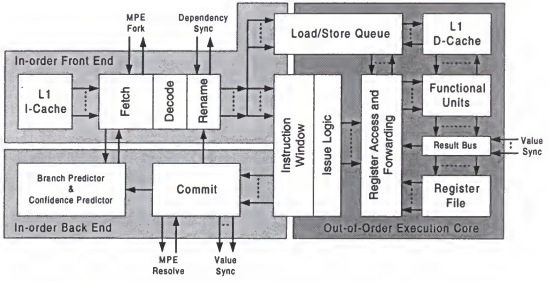


Figure 16. Processor microarchitecture with support for MPE on a CMP

MPE fork. If free resources exist, the primary processor generates a path fork command in response to detecting a low-confidence, conditional branch. The command instructs a single free processor to begin fetching instructions at the non-predicted path. The forked processor inserts a specially-tagged branch instruction into its instruction stream to prevent subsequent instructions from being committed until the branch is resolved by the forking processor.

To allow the forking processor time to determine the address of the non-predicted path, path forks occur at the end of fetch; we assume a 3-cycle fetch stage. With certain

interconnect topologies, alternate processors may be permitted to fork additional alternates.

Dependency Sync. Because alternate processors may fetch instructions that depend on outputs produced prior to the forked branch, the primary processor must communicate a list of register output dependencies. Dependencies are handled by inserting special instructions into the stream that await the specified values from the source processor. Renaming logic [TOM67] enforces read-after-write (RAW) dependencies between fetched instructions and remote dependencies.

Dependency information can be transmitted continuously as instructions are decoded, or all-at-once when a fork occurs. In systems where alternate processors can fork new paths, the alternates must also communicate dependency information.

Value Sync. Maintaining synchronization between processors on the CMP comprises the largest amount of inter-processor bandwidth. This synchronization requires the primary processor to communicate any changes of programmer-visible registers to all other processors on the chip. In the proposed design, only committed register values are synchronized. Therefore, value synchronization always occurs between the primary processor and all other processors. The free and alternate processors can commit register values received from the primary processor.

Though these values may be available earlier, the communication requirements are much lower if registers are synchronized in-order. Out-of-order synchronization requires a mechanism to cancel speculative data values in the case of high-confidence branch mispredictions. Also, simultaneous value communication between alternate processors and the ability for freed processors to "resynchronize" to a path that may have

proceeded along to a different state would be necessary. In-order synchronization maintains a single, global state consisting of register values produced in program order. The MPE performance when committing instructions in-order was found to be on the order of 3-5% less than that achieved when values are propagated as soon as they are known.

MPE Resolve. When the primary processor resolves a correctly-predicted branch that forked an alternate path, the alternate path must be freed. For a mispredicted branch, the primary processor is freed and the alternate becomes the new primary processor. A path resolve command matches the specially-tagged branch instruction in the alternate processor's instruction stream, notifying it of the branch outcome.

If a freed processor has forked any alternate paths, they too must be freed. Freeing a processor is otherwise identical to a branch misprediction in a speculative uniprocessor: uncommitted instructions are discarded while committed values (including those received from the primary processor) are retained.

Supporting these four types of MPE data requires a dedicated interconnect between processors. The bandwidth and latency requirements of this interconnect are explored in Section 4.4. It is assumed that cache traffic is handled on a separate interconnect.

Memory dependencies are handled as a special case of register dependencies. In addition to register output dependencies, *dependency sync* data includes the full instruction (minus the opcode field) for all memory stores. In this manner, alternate and free processors insert remote store instructions into their own load/store queue. The effective address can be computed as soon as the index register is valid. Similarly, the

data to be stored is inserted as soon as it is received in a subsequent *value sync*. In this manner, memory disambiguation, RAW dependencies, and speculative stores can be handled identically to the uniprocessor case with the exception that values may arrive from a remote processor rather than a local functional unit.

Alternately, memory disambiguation structures such as the Address Resolution Buffer (ARB) in [FRA96] or Memory Disambiguation Table (MDT) in [KRI98] could also be used. In these schemes, loads are performed as early as possible and if a dependency is later detected, execution rolls back to a point prior to the load. The proposed approach takes advantage of the lower-latency MPE communication to avoid costly state maintenance and rollbacks which would be required if memory dependencies are detected through the cache hierarchy.

As noted in other studies, the effectiveness of a conventional return-address stack (RAS) is highly reduced when MPE is employed [AHU98]. Instead, a per-path RAS is required to prevent subroutine calls in simultaneously executing paths from polluting the stack. In an MPE-capable CMP, each processor maintains an independent RAS. Subroutine calls and returns are communicated in the same manner as path forks, causing free processors to push or pop the return addresses onto their RAS. Alternate processors buffer these values when executing their own instructions, instead using the RAS for subroutines in the alternate path. When an alternate processor is freed, it recovers the RAS to the state of the primary processor exactly as a speculative processor [JOU97] and then applies the buffered values.

4.2 Simulation Environment

To evaluate the performance of MPE on a CMP, SimpleCMP was augmented with the architectural features and limitations described in this chapter to support MPE. Since simulator speed was not a design goal for the MPE studies, the MPE extensions are limited only to sequential simulations. The MPE extensions significantly reduce the computation-to-communication ratio, making parallel simulation less efficient.

Table 5 shows the key architectural parameters for each individual processor in the simulations, including four levels of processor complexity. The 4-issue variant is modeled after the Alpha 21264 while 8-, 16-, and 32-issue designs were extrapolated by increasing key resources. Since branch mispredictions occur in the commit stage, the misprediction latency is a minimum of 8 cycles, but it can be much longer if the branch must wait for an input dependency before issuing.

A "ones counter" branch confidence predictor [JAC96] is employed to determine whether a branch prediction has high or low confidence. The predictor is indexed by branch address and contains a shift register representing the result of the last eight branch predictions. A set bit represents a correct decision. If two or fewer bits are set, the branch has low confidence and, if resources are available, will cause a forked path to follow both branch directions. This style of confidence predictor was selected because it can be implemented simply in hardware and shows the best performance in other MPE studies [AHU98]. Significant speedup is obtained by using ideal branch confidence prediction that assigns low confidence to every mispredicted branch and high confidence otherwise. Since this effect is well documented [AHU98, KLA98], only realistic confidence prediction is considered in this study.

Table 5. Processor architecture parameters for MPE evaluation

	4-issue	8-issue	16-issue	32-issue
Fetch/Issue/Retire Width	4 instr.	8 instr.	16 instr.	32 instr.
Rename Registers	128	256	512	1024
Integer Issue Queue	64 instr.	128 instr.	256 instr.	512 instr.
Load/Store Issue Queue	32 instr.	64 instr.	128 instr.	256 instr.
Integer ALUs	4	8	16	32
Memory Ports	2	4	8	16
Branch Predictions/cycle	1	2	4	8
Branch Predictor	*Global*: 12-bit history, 4k × 2-bit saturating counters *Local*: 1k × 10-bit history, 1k × 2-bit saturating counters *Select*: 12-bit global history, 4k × 2-bit saturating counters			
Branch Target Buffer	2048 entries, 2-way associative			
Return Address Stack	32 entries			
Minimum Misprediction Latency	8 cycles			
Branch Confidence Predictor	Ones counter, 2048 entries, 8 bits/entry, threshold 2			
L1 I-Cache	64 kB, 2-way associative, 32-byte lines, 1-cycle latency			
L1 D-Cache	64 kB, 2-way associative, 32-byte lines, 1-cycle latency			
Unified L2 Cache	8 MB, 4-way associative, 32-byte lines, 12-cycle latency			
Memory Access Latency	100 cycles, 8 bytes/cycle			

The performance of MPE is gauged on the simulated CMP by executing components of SPECint95 [SPE96] with and without MPE on the processor architectures described above. All benchmarks are compiled using *gcc* 2.6.3 and the options "`-O3 -funroll-loops`". Only the integer benchmarks were simulated since the floating-point components show near-perfect branch prediction accuracies. SPEC95 was selected because simulation times become prohibitive with the larger data sets and longer runtimes of SPEC2000. Also, SPEC2000 is more memory-bound than SPEC95 [HEN00]

whereas this study is intended to focus on the effect of branch mispredictions. Table 6 summarizes the SPECint95 benchmarks and input data sets used in the simulations.

To reduce simulation times, the sampling scheme from [SKA99] is applied. Because the start of each benchmark is not representative of the overall behavior, the benchmarks were simulated using a fast, functional simulation for the specified number of "warmup instructions" shown in Table 6. Only the cache and branch predictors were simulated in this stage. The following 50M instructions were simulated using a fully detailed, trace-driven pipeline simulation. Statistics were gathered only during this 50M-instruction window. The window is selected for each benchmark in such a way as to accurately represent the overall behavior of the program.

Table 6. SPECint95 benchmarks parameters.

Benchmark	Input	Warmup Instr.	Branch Predictor		Confidence Predictor		
			Accuracy	Path Length	Accuracy	Path Length	% Increase
compress	bigtest.in	2576 M	87.12 %	58	76.76 %	250	331 %
gcc	cccpi.i	221 M	86.26 %	47	61.78 %	122	160 %
go	9stone21	926 M	75.40 %	37	80.28 %	185	400 %
ijpeg	vigo.ppm	824 M	87.75 %	161	75.44 %	655	307 %
li	*.lsp	271 M	93.41 %	100	56.45 %	229	129 %
m88ksim	test	26 M	95.36 %	126	53.90 %	272	116 %
perl	scrabbl.pl	601 M	93.13 %	105	50.67 %	213	103 %
vortex	test	2451 M	98.80 %	635	46.22 %	1181	86 %

Table 6 also provides branch and confidence predictor statistics on the baseline 4-issue architecture. The branch predictor accuracy shows the directional prediction accuracy of conditional branches only. The path length refers to the average number of instructions between all mispredicted branches. For the confidence predictor, the accuracy statistic refers to the percentage of conditional branch mispredictions that are detected as low confidence. With unlimited resources, these branches would trigger an

MPE fork and behave as a correctly predicted branch. The effective path length between mispredicted instructions is therefore increased as shown in the table.

In most of the experimental studies that follow, only the performance results of the *gcc*, *go*, *ijpeg*, and *vortex* benchmarks are examined. The *go* and *vortex* benchmarks represent the largest and smallest path-length increases and hence the best- and worst-case MPE performance, respectively. The *gcc* benchmark represents a middle level of MPE performance while *ijpeg* is included because it has the highest uniprocessor IPC. In addition to these four, the average performance of all eight SPECint95 benchmarks is also provided.

4.3 Architectural Requirements for Efficient MPE

In this section, the impact of several architectural elements on MPE performance is considered. The results presented here assume an ideal interconnect with unlimited bandwidth, single-cycle latency, and crossbar connectivity. The intent of these experiments is to narrow the design space for MPE by establishing the importance of CMP size and complexity, on-chip cache hierarchy and prediction logic, and processor-path allocation by studying idealized systems. Limited interconnect capacity and practical performance is explored in subsequent sections.

4.3.1 Processor Count and Complexity

One of the most important design decisions regarding MPE on a CMP is determining the minimal resources required to yield effective results. In the case of MPE, the goal is to provide speedup on sequential programs. Figure 17 shows the relative improvement in instructions per cycle (IPC) for CMPs of varying sizes and issue widths when compared to a uniprocessor of the same issue width.

The first trend evidenced by Figure 17 is that *go* shows the largest increase in IPC using MPE while *vortex* shows the least (note the different y-axis scales). This result is in line with the benchmarks' branch prediction accuracy and matches results found elsewhere [AHU98, WAL98].

The effect of processor count demonstrates several interesting trends. First, for 4- and 8-issue processors, most of the benchmarks yield diminished returns for more than 8 processors. At 16- and 32-issue, a 16-processor design is preferable. Due to its poor branch prediction accuracy, *go* continues to show marked improvement for up to 32 processors on all but the 4-issue design.

Perhaps the most interesting results from Figure 17 show that MPE yields a larger increase in performance on wider-issue designs, even when the processor count is held constant. This trend can be attributed to the fact that wider processors must fetch more instructions on each speculative path prior to resolving the branch. Therefore, a misprediction results in a larger performance penalty.

The results between different issue widths cannot be compared in an absolute sense because the baseline IPC is different. Also, many variables regarding issue width scaling, such as clock frequency, have not been taken into account in the configurations provided in Table 5. However, the wide-issue speedup demonstrated here is conservative. Though constant misprediction latency is assumed, the wider-issue designs would likely have a longer pipeline, hence larger misprediction latency. Other MPE studies have demonstrated that larger misprediction latencies improve the performance of MPE [WAL98], therefore it is likely that the IPC increase for wide-issue processors would actually be larger than shown in Figure 17.

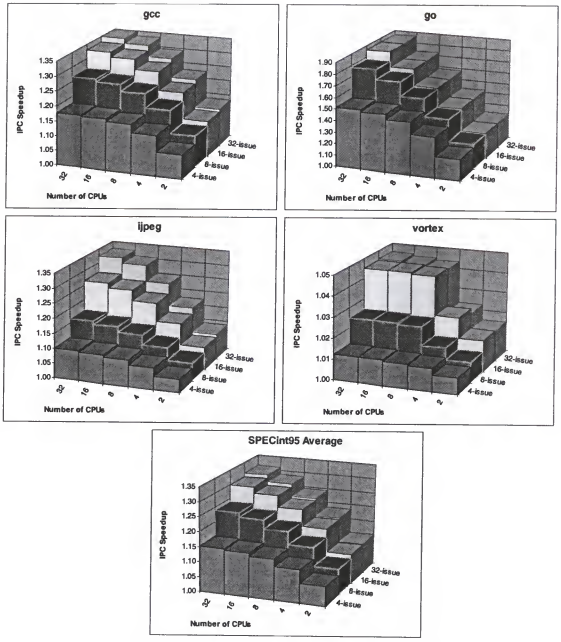


Figure 17. Effect of CPU count and complexity

4.3.2 Cache Hierarchy and Prediction Logic

When a misprediction is detected by the primary processor for a branch that also forked an alternate path, the processor following the alternate path becomes the new primary processor. In this manner, the assignment of primary processor continuously migrates among all processors in the CMP. If there are a large number of processors,

execution may not return to the original primary processor for many instructions. Structures that depend on locality of accesses—namely cache and branch prediction—will not operate effectively if they only see a subset of the instruction stream.

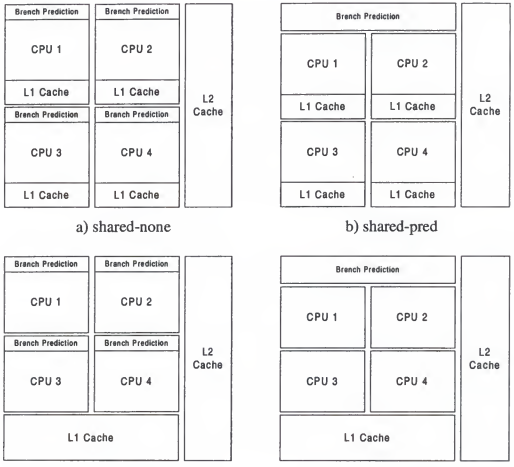


Figure 18. Alternative architectures for a CMP

Figure 18 shows four alternative architectures for a CMP. The *shared-both* configuration of Figure 18d is ideal for MPE since each processor shares a common L1 cache and branch predictor. However, the *shared-none* design in Figure 18a is preferable for implementation since each processor has low-latency access to its own L1 cache and branch prediction hardware. Figures 2b and 2c represent hybrid designs of *shared-pred*

and *shared-L1* with either a shared branch predictor or a shared L1 cache, respectively. In all cases, a shared L2 cache hides main memory accesses with a tolerable amount of on-chip communication latency.

Because of its important effect on performance, it is assumed that a per-path RAS is maintained even in the *shared-both* and *shared-pred* architectures.

Reflective L1 cache

Since the physical design of *shared-none* is ideal for implementation but less suited to MPE performance, a mechanism for a *reflective* L1 cache is proposed to provide the logical equivalent of *shared-L1* with the physical design of *shared-none*. To mimic the effect of a shared L1 cache, all cache lines requested by the primary processor are also loaded into the L1 cache of each free processor. To achieve this goal, the L2 cache maintains a 2-bit state table for each processor indicating whether a processor is the *primary*, an *alternate*, or *free*.

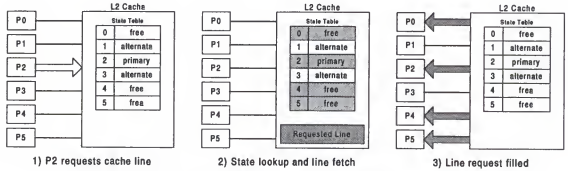


Figure 19. Example of a reflective cache architecture

Figure 19 shows an example of such an implementation on a 6-processor CMP. The state table indicates that the primary processor is P2, P1 and P3 are alternates, while P0, P4, and P5 are free. In step 1, P2 encounters an L1 cache miss and requests the data from the L2 cache. In step 2, the L2 cache simultaneously checks its state table and

fetches the requested line from the cache or main memory. Step 3 shows the fetched cache line being broadcast to the requesting processor as well as all free processors. Since a free processor does not execute instructions, there will never be a conflict requiring simultaneous transfer of two cache lines to a single processor.

This design requires each processor to notify the L2 cache of a change in state whenever an MPE fork is created or resolved. Since the primary processor must maintain its own state table, an alternative implementation could require the primary processor to identify all free processors along with its L2 cache requests.

Performance impact

To gauge the effect of cache and branch prediction sharing, CMP architectures exhibiting the *shared-none*, *shared-pred*, *shared-L1*, and *shared-both* architectures were simulated. A fifth alternative using the *reflective-L1* design with independent branch prediction logic was also simulated. The resulting increases in IPC over a uniprocessor are shown in Figure 20. All designs used an 8-processor, 8-issue CMP, though similar trends were observed for designs with other processor counts and issue capabilities.

While some benchmarks such as *ijpeg* and *vortex* exhibit little difference among the five configurations, the general trend is that *shared-both* performs best while *shared-none* performs worst. The *shared-L1* results are closer to ideal than the *shared-pred* results, particularly for *gcc*. Therefore, the reduced cache locality has a more significant effect on performance than decreased branch prediction accuracy. This result is unsurprising since MPE is designed to reduce the number of branch mispredictions, mitigating the loss in accuracy.

Figure 20 demonstrates that a *reflective-L1* design, though not as effective as a single, shared cache, provides much of the benefit while still allowing distributed caches.

On the average, the *reflective-L1* provides a 17.0% speedup compared to only 12.6% speedup with a *shared-none* design. Though *shared-both* can provide 19.0% speedup, the implementation disadvantages of such an approach make the reflective cache a clear winner.

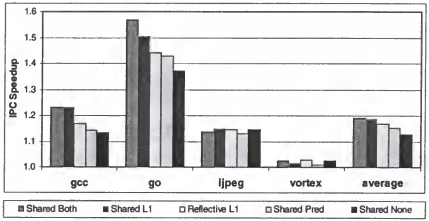


Figure 20. Effect of cache and branch prediction sharing

4.3.3 Allocation Strategies

Allowing alternate-path processors to fork additional alternates requires a topology that can support simultaneous transactions. In this section, three allocation strategies that limit the degree to which alternate processors can fork new paths are considered: *primary only*, *fork-N*, and *eager*.

Primary Only. In this strategy, only the primary processor (following the predicted path) can fork alternates. This policy is similar to the Single Path (SP) strategy in [UHT95]. Such a pattern is well suited to a bus topology since no communication between alternates is necessary.

Fork-N. Each processor is allowed to fork *N* alternates, regardless of whether it is the primary processor, provided there is a free processor. This approach is similar to allowing nearest-neighbor forking on a ring or torus topology without contention.

Eager. All processors can fork alternates as long as free resources remain. In [UHT95], this approach is called Eager Execution (EE). Supporting this policy requires crossbar connectivity on a CMP. Note that *eager* allocation is equivalent to *fork-7* in an 8-processor CMP.

Figure 21 shows the increase in IPC when MPE is applied to an 8-processor, 8-issue CMP using these allocation strategies. The communication bandwidth and latency is assumed to be ideal and a shared-L1 architecture is used.

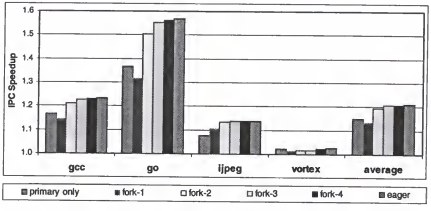


Figure 21. Effect of fork tree limitations

The results clearly demonstrate an advantage to allowing alternate processors to fork new paths. The *primary only* scheme provides an average of 15.1% speedup while the *eager* scheme shows a 20.9% average speedup. Allocating only one path fork per processor yields the lowest average performance increase at 13.5%. However, for all but the *go* benchmark, a *fork-2* approach is nearly as effective as *eager* but requires a much simpler topology.

One reason for the low performance of both *primary only* and *fork-1* allocation strategies is that the processors are underutilized. Both approaches require seven outstanding low-confidence branches along a single path to fully utilize all processors, a

condition which rarely occurs. The exponential effect of *fork-2* is sufficient to occupy all processors with three outstanding branches in each possible path.

In other MPE studies, more sophisticated allocation techniques are explored which consider such factors as cumulative branch prediction probabilities and relative confidence levels to determine the optimum set of paths to explore with limited resources [UHT95, KLA98]. Though some of these techniques provide greater performance than the *eager* approach, they rely on centralized arbitration to consider all outstanding branches. In this study, the schemes are limited to consider only those that can function with the distributed control inherent in a CMP.

In practice, the allocation strategy will be determined by the topology of the interconnect between processors on the CMP. Section 4.5 will consider the effect of several interconnect topologies on allocation policy.

4.4 Communication Requirements for MPE

Because of the increasing dominance of wiring delays in large-scale integrated circuits, the interconnect between processors of a CMP will necessarily have limited bandwidth and non-negligible latency. The ability of MPE to function on a CMP depends on the match of communication requirements to interconnect capacity. In this section, the requirements for the on-chip interconnect to support MPE are evaluated.

4.4.1 Bandwidth Requirements

The bandwidth required to support MPE on a CMP depends on relative amounts of the four types of synchronization data discussed in Section 4.1: *MPE fork*, *dependency sync*, *value sync*, and *MPE resolve*. Each one will be treated in turn by using statistics gathered from an 8-processor CMP with 8-issue processors.

For these experiments, per-processor bandwidth is the desired metric. Therefore, only the primary processor is permitted to fork new paths. This represents a worst-case scenario for *MPE fork* and *MPE resolve* operations because all of these commands come from a single source. The bandwidth required for *dependency sync* data was found to be similar for all topologies. *Value sync* bandwidth applies only to the primary processor and hence is independent of topology.

Fork and resolve operations

The required bandwidth for *MPE fork* commands depends upon the rate at which low-confidence branches are encountered. The effective rate is reduced when there are no free processors with which to fork. The rate of *MPE resolve* commands is identical to the fork rate.

Table 7. MPE fork requirements for an 8-processor, 8-issue CMP

Benchmark	Fraction of Cycles	
	Low-conf Branch	MPE Path Fork
compress	13.8 %	13.5 %
gcc	16.8 %	13.6 %
go	24.3 %	23.2 %
ijpeg	6.3 %	6.3 %
li	10.0 %	9.9 %
m88ksim	6.6 %	6.4 %
perl	8.6 %	8.5 %
vortex	1.9 %	1.9 %
Average	11.0 %	10.4 %

Table 7 indicates the relative frequency of low-confidence branches and *MPE fork* commands for each of the SPECint95 benchmarks. The statistics are taken from the migrating primary processor in an 8-issue, 8-processor CMP. Overall, an *MPE fork* is generated in only 10.4% of all clock cycles, though *go* generates alternate paths at over twice the average rate. The data indicate that even if resources were sufficient to allow

forking on every low-confidence branch, the number of forks would not increase by a significant amount.

The results indicate that supporting a single *MPE fork* and *MPE resolve* command per cycle is sufficient for an 8-issue design. The amount of data required for an *MPE fork* is somewhat more than that of an *MPE resolve* command. Both types must identify a destination processor, requiring 3 bits for an 8-processor CMP. Each *MPE fork* command provides the full 32-bit address at which to start fetching instructions. An *MPE resolve* command merely contains a single bit to indicate whether the destination should cancel its speculative state or become the new primary processor.

Dependency information

When an alternate path is forked, the new path must have knowledge of which register values have unresolved output dependencies in the forking processor and what memory values are to be changed. Two approaches to register *dependency sync* can be taken: continuous, per-cycle dependency information and fork-time dependency communication. The number of output dependencies required on a per-cycle and fork-time basis are shown in Table 8.

On average, about 4 register dependencies are generated per clock cycle while only slightly more (i.e., 4.5) are outstanding at the time of an *MPE fork*. The reason for this effect is that a relatively small set of registers accounts for the majority of all writes, resulting in a large number of write-after-write (WAW) dependencies. Figure 22 illustrates the magnitude of this effect averaged over all 8 SPEC95 integer benchmarks. A single register, R2 in this case, is the destination for 37% of all register writebacks. Together with R3, the two registers account for almost 50% of all writebacks, while over 90% are made to a set of only 12 registers.

Table 8. Dependency synchronization requirements for an 8-issue processor

Benchmark	Output Dependencies/Cycle		Dependencies at Fork Time	Stores/Cycle	Stores at Fork Time
	All	New			
compress	3.204	1.702	4.372	0.479	1.230
gcc	6.074	1.563	4.018	0.357	0.918
go	6.651	2.203	5.506	0.279	0.692
ijpeg	4.975	1.271	5.203	0.351	1.436
li	3.009	1.484	4.236	0.483	1.380
m88ksim	3.318	1.104	4.267	0.348	1.343
perl	3.021	1.433	4.162	0.663	1.924
vortex	2.474	1.628	4.759	0.734	2.146
Average	4.091	1.549	4.565	0.462	1.361

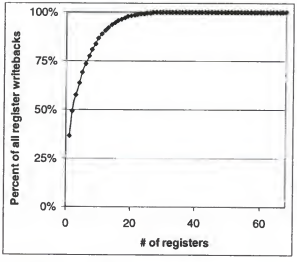


Figure 22. Register writeback distribution for SPECint95

The frequent WAW dependencies can be used to lower the rate of continuous *dependency sync* information. If a writeback occurs to a given register multiple times before a forking branch, only the latest value need be transmitted to a processor working on the alternate path. Table 8 demonstrates that only about 1.5 "new" dependencies are generated per cycle.

One significant disadvantage to the continuous approach is that a free processor must receive the *dependency sync* commands from any processor that may start an

alternate path on it. This requirement is not necessary if the dependency information is communicated along with the fork command. If only the primary processor can start alternates, then continuous *dependency sync* is easily achieved.

Communicating the dependency information in either approach is accomplished by simply listing the registers that are expected to be changed. For the architecture used in this study, 7 bits per dependency are required, though this could be lowered to as few as 5 bits if floating-point and general-purpose registers are grouped.

Memory dependencies are enforced by passing the details of each store instruction (i.e., index register, value register, and immediate offset), requiring up to 32-bits of data per store. Table 8 shows that, on average, 0.46 stores are fetched per clock cycle on an 8-issue processor. The worst case, *vortex*, still averages less than one store per cycle at 0.73. Since optimized code rarely produces temporally-adjacent stores which overlap with previous stores to the same address, all stores are communicated. If fork-time dependency communication is employed, an average of about 1.4 stores will be outstanding, though *vortex* produces over 2.1 stores per fork operation.

Value synchronization

Of all the synchronization data types, the *value sync* data has the highest potential requirements. In an 8-issue processor using the 32-bit MIPS instruction set, up to 16 register values can be updated each cycle with double-word instructions (in practice, this may be limited by the number of register file input ports). Supporting this amount of interprocessor bandwidth would be prohibitive. Fortunately, the actual *value sync* bandwidth is limited by three factors:

1. average IPC
2. ratio of instructions which actually write to a register
3. percentage of WAW dependencies

The first two factors are self-explanatory. The third factor accounts for the same register reuse effect that limits the number of *dependency sync* transactions.

Table 9 quantifies these factors on SPECint95 using an 8-issue uniprocessor. Due to cache accesses, branch mispredictions, and true dependencies, the IPC is significantly lower than 8—about 2.6 on average. Even the highest benchmark, *ijpeg*, exhibits an IPC of only 3.8.

Table 9. Value synchronization requirements for an 8-issue processor

Benchmark	IPC	All Register Writes		Latest Write Only	
		writes/inst	words/cycle	% overlap	words/cycle
compress	1.921	0.641	1.231	34.1 %	0.811
gcc	1.822	0.704	1.283	57.5 %	0.545
go	1.879	0.784	1.473	49.2 %	0.749
ijpeg	3.808	0.881	3.355	70.9 %	0.977
li	2.398	0.640	1.535	40.6 %	0.912
m88ksim	2.798	0.706	1.975	62.2 %	0.747
perl	2.633	0.645	1.698	44.6 %	0.941
vortex	3.401	0.645	2.194	30.3 %	1.530
Average	2.583	0.694	1.843	51.1 %	0.902

The ratio of instructions that write to a register is quantified by the *writes/inst* metric. This value accounts for instructions that do not write to a register at all as well as instructions that write multiple registers. On average, there are about 0.7 register writes per instruction, though *ijpeg* has the largest ratio at almost 0.9. When this metric is combined with the IPC, the expected number of *words/cycle* is found. As Table 9 illustrates, *ijpeg* requires the most bandwidth at 3.4 words per cycle while the average is only 1.8.

When per-path WAW dependencies are taken into consideration, the required bandwidth is dramatically lowered. In Table 9, the *% overlap* metric refers to the percentage of register writebacks that do not need to be transmitted due to a later write before the next path fork. Over 51% of register writes were found to overlap, though individual benchmarks varied from 30% to 71%. This effect lowers the required bandwidth to less than 1 word/cycle. While *ijpeg* shows the highest IPC and highest ratio of register writes per instruction, it also has the largest overlap factor and therefore requires only slightly above average bandwidth. With all factors considered, *vortex* requires the most bandwidth at just over 1.5 words/cycle.

The amount of data each *value sync* transaction contains is equal to the register word size plus a tag to identify the destination register. In the simulated system, the word size is 32-bits and a 7-bit tag uniquely identifies the destination.

MPE performance with limited bandwidth

In order to allow the CMP to function in the presence of limited bandwidth, the commit phase is modified to stop committing instructions when the bandwidth is exceeded. The number of *MPE fork* and *MPE resolve* transactions is fixed to one per cycle. When more than one *MPE fork* is required per cycle, only the first is performed while subsequent branches execute normally. If multiple *MPE resolve* commands are encountered in a single cycle, instruction commit stalls. Fork-time *dependency sync* is employed with up to 5 register dependencies and one memory dependency per cycle. Extra dependencies are queued and sent in subsequent cycles with instruction fetch stalling when the queue fills. A queue with room for 15 register dependencies and 3 store instructions was found to limit the stall frequency to have negligible impact on performance.

These minimal limitations on *MPE fork*, *MPE resolve*, and *dependency sync* were found to have an insignificant effect on performance. Limiting the *value sync* bandwidth, however, does affect MPE performance as demonstrated in Figure 23. An 8-processor system is assumed where the *value sync* bandwidth is varied from 1 to 32 words/cycle. The bandwidth refers to the number of register values that can be transmitted and does not include extra bits to identify the destination register. Results are provided for 4-, 8-, 16-, and 32-issue processors.

The results in Figure 23 indicate that for all benchmarks, the 4- and 8-issue CMPs achieve over 98% of the peak IPC with as little as two words of value sync bandwidth. For *go* and *gcc*, 97% of peak performance is reached with this amount of bandwidth even for 16- and 32-issue processors. Due to their higher requirements, *ijpeg* and *vortex* respectively achieve only 88% and 94% of their peak performance with 16-issue processors and only two words of bandwidth. However, for these two benchmarks and the SPECint95 average, a bandwidth of four words is sufficient to provide 99% of the peak performance.

Though this data show the bandwidth requirements to be substantially lower than the theoretical maximum, the required amount is still somewhat higher than the register overlapping statistics in Table 9 might indicate. It was found that register writes to non-overlapping values frequently occur in bursts, such as when values are restored from the stack. Because instruction commit is stalled when the *value sync* bandwidth is exceeded, these bursts can significantly impact performance if they are not drained quickly. Maintaining a separate queue for value synchronization commands could help alleviate this problem.

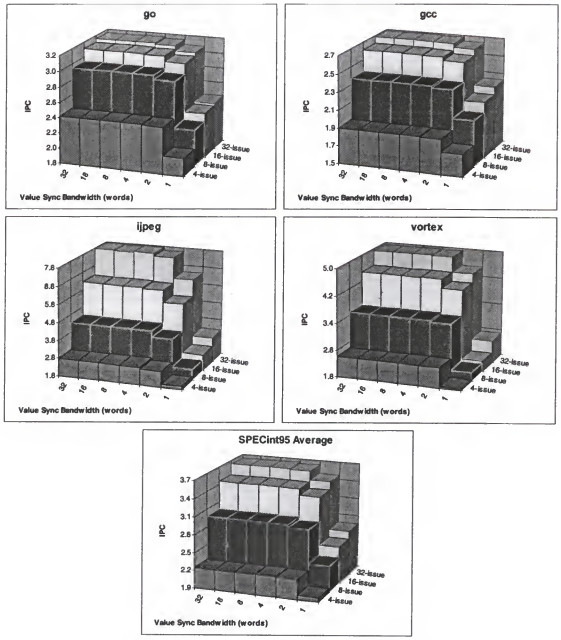


Figure 23. Effect of limited value sync capacity

4.4.2 Latency Requirements

In order for MPE to provide speedup, the latency of interprocessor communication must be less than the average branch misprediction latency. If the correct direction of a branch can be determined on the primary processor in less time than it would take to start an alternate, it is more efficient for the primary processor to

speculate all branches in the conventional manner. Other factors, such as the in-order synchronization penalty and reduced effectiveness of the branch prediction, require that the interprocessor latency be somewhat less than the misprediction latency. In this section, the requirements for interprocessor latency are quantified.

MPE performance with increased latency

In Figure 24, the increase in IPC when using MPE on an 8-processor, 8-issue CMP with crossbar connectivity and unlimited bandwidth is plotted for interprocessor latencies of 1 to 16 cycles. The minimum misprediction latency was also varied from 8 cycles to 24 cycles. The IPC speedup indicates the improvement over a uniprocessor with the same mispredict latency.

As expected, the combination of low interprocessor communication latency and high mispredict latency yields the best results while high communication latency and low mispredict latency performs disastrously. On the average, there is a significant advantage to having an interprocessor latency of 4 cycles or less with an 8-cycle mispredict latency. For mispredict latencies of 12 cycles or more, greater than 10% average speedup is attainable even with 8-cycle communication. A communication latency of 16 cycles becomes feasible with a 24-cycle mispredict latency.

In general, the data suggest that MPE will provide respectable speedup for interprocessor latencies which are between one-half and three-quarters of the mispredict penalty. At higher interprocessor communication latencies, MPE is not beneficial while at lower latencies, the performance of MPE increases drastically.

It is important to note that the actual IPC is lower for larger mispredict latencies. The purpose of these charts is to demonstrate the limits of MPE given a CMP where the mispredict and interprocessor communication latencies are set by some other criteria,

such as clock frequency goals. Indeed, an increased clock frequency associated with longer pipeline depths may offset the decrease in IPC. The IPC speedup demonstrated in Figure 24 can be achieved independently of clock frequency.

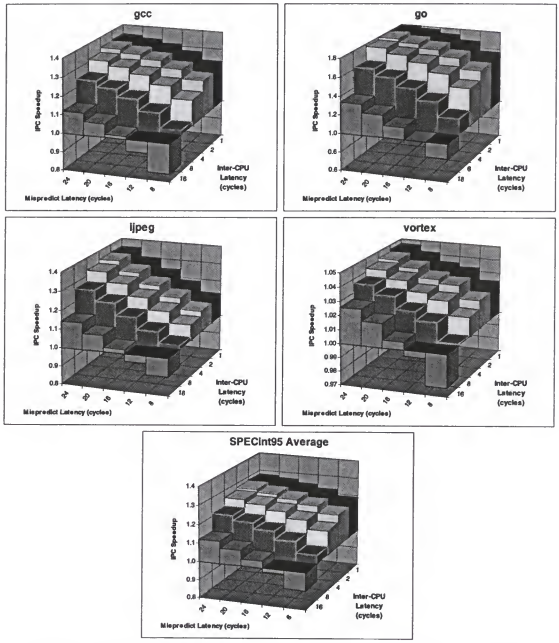


Figure 24. Effect of interprocessor communication and misprediction latencies

4.5 Practical CMP Implementations for MPE

In this section, all of the components from in Sections 4.3 and 4.4 are combined in order to ascertain the performance of MPE on practical hardware. MPE has been shown to be a communication-limited exercise and as such, before considering the CMP as a viable architecture for MPE, communication and integration issues must be addressed. In this section, a theoretical 8-processor CMP featuring aggressive, 8-issue processors manufactured using a 0.1μm fabrication process is targeted for MPE. First, the feasibility of such a design will be established and alternative interconnects considered. Then, simulative analysis is provided to demonstrate the performance of MPE on these realizable systems.

4.5.1 CMP Implementation Considerations

The exploration of MPE performance on a CMP configuration of not-yet-realized processors necessitates transistor count estimates to determine integration limits. As a baseline, the Alpha 21264 microprocessor, which harbors many similarities to the 4-issue processors used in this study, is considered. This processor consists of 15 million transistors, approximately 9 million of which comprise the core logic. The 64KB L1 instruction and data caches consume the remaining 6 million transistors [KES98].

Processor architectures are widely diverse, so while no rigorous approach to scaling exists, a common heuristic assumes transistor counts increase as a quadratic function of issue width [COD99, LIP97, OLU96]. Applying this method, an 8-issue core would require approximately 36 million transistors. The L1 cache sizes remain 64KB, and a multi-banked approach is assumed to provide increased fetch width without a significant increase in area, resulting in a total per-processor transistor count of approximately 42 million.

Throughout these experiments, the simulated L2 cache size has been held constant at 8MB regardless of CPU count or issue width. Assuming six transistors per SRAM cell, this memory requires approximately 500 million storage transistors and an estimated additional 100 million transistors for tags and control. Together with 8 processors, a total of 946 million transistors are required. As the 0.1μm feature size will likely herald the advent of gigascale integration [MAT97], additional transistors and chip area will be available to implement the interconnection systems required for MPE.

The latency for an interprocessor communication will depend on the proximity of the source processor to the destination. A processor with 42 million transistors comprises roughly 4.2% of a billion-transistor chip. Matzke [MAT97] argues that in a billion-transistor, 0.1μm process chip, 16% of the chip is reachable in one clock cycle, indicating that any two adjacent processors can communicate with each other in one cycle. Similarly, the entire die may be spanned in 8 cycles.

The actual interprocessor communication latency will be dependent on topology. Figure 25 shows five potential topologies for the MPE interconnect: *bus*, *ring*, *mesh*, *torus*, and *crossbar*. The ring and torus configurations can include either uni- or bi-directional links (note that for the simple 8-node torus in Figure 25, only the horizontal links can be unidirectional). Table 10 compares the five topologies in terms of estimated latencies and restrictions on the fork tree.

It is assumed that the bus and crossbar networks would have the highest latency at 8 cycles due to the need to span a large portion of the chip. The ring and mesh networks would allow single-cycle communication with neighboring nodes. For the torus, a folded

design requires longer links to accommodate the wrap-around signal, so a 2-cycle link latency is allowed.

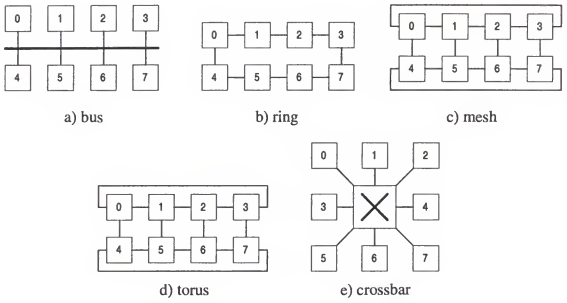


Figure 25. MPE network topologies for an 8-processor CMP

Table 10. Comparison of topology alternatives for MPE on an 8-processor CMP

Topology	Links	Latency		Maximum Path Forks	
		Per Link	Maximum	Primary	Alternate
bus	bi-directional	8 cycles	8 cycles	7	0
ring	unidirectional	1 cycle	7 cycles	1	1
ring	bi-directional	1 cycle	4 cycles	2	1
mesh	bi-directional	1 cycle	4 cycles	2 or 3	1 or 2
torus	unidirectional	2 cycles	8 cycles	2	2
torus	bi-directional	2 cycles	6 cycles	3	2
crossbar	bi-directional	8 cycles	8 cycles	7	6

Though all nodes must eventually receive the *value sync* data from the primary processor, a pipelined approach is employed in the ring, mesh, and torus networks wherein nodes closest to the primary processor receive the data sooner than those which

are farther away. The worst-case latency is provided in Table 10 to indicate when the farthest node receives this data.

Finally, the topology places restrictions on which processors can start alternate paths. Since the bus network does not support concurrent communication, only the primary processor can start alternate paths. For the ring, mesh, and torus networks, any processor can start an alternate path on a neighboring processor, provided that it is free. The crossbar allows any processor to start an alternate path on a free processor. If multiple processors contend for a single free processor in the same cycle, the free processor selects the lowest numbered processor and notifies the others that the fork request has failed.

Table 10 shows the maximum number of new paths that the primary and alternate processors may fork provided there is no contention. Note that for a mesh, this number depends on whether the processor is located on an edge or in a corner.

4.5.2 Practical MPE Performance

In this section, the performance of MPE is presented when all practical limitations are provided. An 8-processor, 8-issue design is used with the reflective L1 cache architecture and two words per cycle of *value sync* bandwidth. The latency of the interconnect depends on the topology as shown in Table 10.

The topology also determines the allocation strategy. A bus topology results in *primary only* allocation. A unidirectional ring is equivalent to the *fork-1* strategy. The bi-directional ring is similar to *fork-1* but the primary processor can fork two paths. A unidirectional torus is similar to *fork-2* except that there is contention for free processors between different paths. The mesh and bi-directional torus also approximate *fork-2* but

allow the primary processor to start three alternates. A crossbar allows *eager* path allocation.

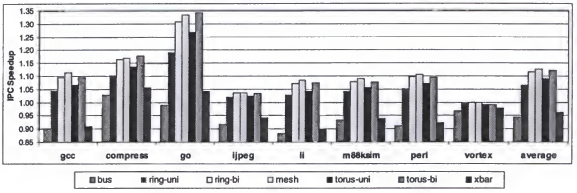


Figure 26. MPE performance with practical topologies and latencies

Figure 26 compares the speedup obtained through MPE for each topology over an 8-issue uniprocessor. Results for all eight SPECint95 benchmarks are provided along with the average speedup.

The results show that the high latency of the bus and crossbar topologies negate any performance increase from MPE despite the fact that the crossbar scheme provides the best-performing allocation scheme. The low latency afforded by a unidirectional ring topology allows a 6.7% average speedup with almost 19% on *go*. Allowing the primary processor a second path fork via a bi-directional ring increases the average speedup to 11.6%. This second fork allows the remaining processors to be occupied with half the number of low-confidence branches per path.

The highest overall speedup is provided by using a mesh interconnect—12.7% on average, 33.5% on *go*. The higher latency of the torus interconnects results in decreased speedup with the exception of the bi-directional torus on the *compress* and *go* benchmarks. Though not shown, a bi-directional torus slightly outperforms the mesh

across the board if single-cycle latencies are achievable. Also, for a large number of processors, the bi-directional torus has additional advantages over a mesh due to the fact that there are no edge nodes.

The worst performing benchmark, *vortex*, shows no speedup whatsoever for the ring and torus topologies. All other topologies result in a slight slowdown as the inherently high branch prediction accuracy affords little opportunity for MPE to recoup the performance lost due to reduced cache hit rate and branch prediction accuracy.

4.6 Related Research

Although modern correlating [YEH93] and combining [MCF93] branch predictors can yield average prediction accuracies in excess of 90%, future processors using multiple branch predictors or trace caches [ROT96] are expected to fetch past multiple conditional branches on every clock cycle. With long pipelines, the number of in-flight branches quickly increases the probability of a misprediction. Many techniques to reduce the number of conditional branches have been proposed to improve the performance of modern ILP processors in light of imperfect branch prediction accuracy.

Conditional [YEA96] and predicated [AUG98, JOH96] instructions use the compiler to reduce conditional branches. Both techniques rely on special tags to allow an instruction to execute but discard the result if an associated condition is not met. An advantage to this approach is that a misprediction causes only some instructions to be discarded rather than generating a complete pipeline flush. However, compiler support is required to identify paths that are short enough to use such a scheme. Instruction reuse [SOD97] is another method to reduce the pipeline-flush penalty that does not require compiler support but still relies on short path lengths.

For longer path lengths, techniques to explore both the predicted and the non-predicted path have been developed. One of the first such MPE techniques is Disjoint Eager Execution (DEE) [UHT95]. DEE employs a novel processor architecture to track the simultaneous execution of instructions from multiple conditional paths in a program. In DEE, the compiler provides static prediction probabilities to determine which branches should execute both possible paths. Selective Eager Execution (SEE) [KLA98] extends DEE to a more traditional superscalar processor and employs confidence prediction. Special tags associate each instruction with a path, allowing mispredictions to flush only those instructions that are on an incorrect path.

Other approaches to MPE use the hardware of a simultaneous multithreaded (SMT) processor to track different paths [AHU98, WAL98]. When distinct threads of execution are unable to fully use all of the functional units, one or more alternate paths are generated until the hardware resources are fully utilized. Efficient partitioning of instruction fetch bandwidth on an SMT to support MPE is explored in [KLA99]. However, these studies do not consider the effect of signaling delays on future designs.

For example, large-scale integrated circuits may not provide optimal clock frequencies for monolithic designs such as superscalar processors [MAT97], making SEE less desirable. Similarly, architectures based on multithreaded processors do not show good signal locality in all stages. Path creation requires the working set of register values for all threads be accessible by all other threads (so that two alternate paths can read values from the parent path) or that the working set of registers is duplicated so that each path has its own copy. Either implementation may be impractical if the size of the register files is limited by the amount of chip area that can be traversed in a single clock

cycle. Thus, while MPE has been shown to provide a modest performance gain on superscalar and multithreaded hardware, the technique may not be optimal on future circuit implementations with dominant signaling delays [KEC98].

4.7 Summary

Future processor architectures will leverage the integration of multiple CPUs on a single chip. The results in this chapter demonstrate the potential for increased performance of sequential programs by using MPE on a CMP. An average speedup of 12.7% on SPECint95 was measured, with a 33.5% speedup on *go* due to its low branch prediction accuracy. This level of performance is achievable on an 8-processor, 8-issue CMP using modern branch prediction techniques and realistic branch confidence prediction.

The achieved speedup is comparable to that achieved on other MPE studies using monolithic designs such as wide-issue superscalar and SMT processors. It is likely that a CMP will permit higher clock frequencies and shorter design time. Also, it is well-established that increased ILP through issue width scaling alone has reached the point of diminished returns [WAL93]. MPE provides a means to increase ILP beyond issue-width scaling and independently of clock frequency.

Additionally, it has been shown that even larger speedups may be obtained on more complex systems. MPE shows increased performance with both wider issue widths and longer misprediction latencies. Since the historic trend in processor design has been to increase both issue width and pipeline length, it is likely that MPE will be an even more attractive solution in the future.

A *reflective-L1* architecture was introduced to allow a CMP to maintain independent L1 caches while preserving cache locality and hit rate for effective MPE performance. It was demonstrated that the required interprocessor communication bandwidth can be manageably limited without significantly impacting performance. Furthermore, CMP-based MPE is very conducive to topologies that limit communication to adjacent processors, allowing simpler interconnects and low latencies.

One avenue for future research would be to explore the effect of clock frequency on future designs. Since wider-issue processors would likely require a lower clock rate than the simpler processors, the performance benefit of MPE may be even more pronounced. The performance of MPE itself can be improved in a number of ways. More sophisticated confidence prediction can be used to improve overall performance. With compiler support, instructions whose values will not be used past a branch instruction can be flagged to indicate that the value need not be communicated to alternate processors, further reducing the bandwidth requirements. The reduction in performance due to in-order value synchronization can be partially reclaimed by applying value prediction [SAZ97]. This technique would enable an alternate processor to speculatively execute instructions which depend on data produced by a remote processor.

With architectural techniques such as MPE and innovative implementations such as CMPs, it should be possible to continue the trend of simultaneous increases in processor IPC and clock frequency. In this way, the rapid advance in processor performance can continue.

CHAPTER 5
APPLICATION TO RELATED CMP RESEARCH

While the MPE technique described in Chapter 4 offers a mechanism by which unmodified, sequential code can use the hardware of a CMP for increased performance, the highest performance gains are obtained from truly parallel techniques where each processor works on a valid instruction path. Many other studies have applied such techniques to a CMP, requiring modification of applications or explicit programmer intervention to take advantage of the parallelism inherent in CMP architectures.

In this chapter, the hardware requirements of the MPE technique presented in the previous chapter are compared with the required support for several proposed CMP-based parallelization studies. The goal of these comparisons is to illustrate what additional architectural features are necessary to support both MPE and each parallelization approach. Also, where appropriate, a discussion of how these additional architectural features can benefit MPE is also included.

5.1 Approaches to Application Parallelization

When parallelizing code, several granularities are commonly employed. Figure 27 illustrates the most common levels and how such parallelism is traditionally uncovered. The granularity of parallelism is defined by the average number of instructions in each parallel path. The figure also shows an approximate instruction length for each level, though there are often exceptions with more or less instructions for a given level of granularity.

At the coarsest granularity is program-level parallelism. Such parallelism is found in a multitasking or multiuser operating system where multiple, independent programs share system resources. Program-level parallelism is often used to describe true parallelization where a programmer manually divides a single task into distinctly separate processes that can run simultaneously. Typically, separate instances of the parallelized code are run on multiple nodes of a distributed system, communicating over a network. This level of parallelism was employed in the parallel CMP simulator presented in Chapter 3.

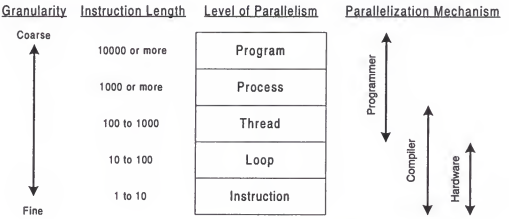


Figure 27. Levels of granularity in program parallelization

Process-level parallelism is an extension of programmer-generated, program-level parallelism where each parallel process is run on a single multiprocessor system such as a symmetric multiprocessor (SMP). The processes are often started automatically using the UNIX `fork()` system call or a similar mechanism, although each process retains an independent address range. Inter-process communication occurs through the operating system via pipes, mapped shared-memory regions, or shared files.

Thread-level parallelism allows a program running on a multiprocessor to spawn multiple tasks that share a common address space. Such threads can be used to complete a small task such as a function call in parallel with the continuation of the regular body of the program. While most thread-level parallelism in the past has been programmer-generated, much research has been put into parallelizing compilers and architectures for exposing thread-level parallelism automatically. The multiscalar architecture [SOH95] described later in this chapter is one such example.

Loop-level parallelism is a relatively new concept in parallel programming. At this level of granularity, compiler and hardware support is used to identify loops where multiple iterations of the loop can be overlapped and executed in parallel. The multiscalar approach and the Hydra CMP [OPL97] each provide support for exploiting loop-level parallelism in programs.

The finest granularity is instruction-level parallelism (ILP). Such parallelism is identified by the processor hardware, although compiler support is usually necessary to arrange the code optimally. ILP typically refers to the ability of a superscalar processor to execute multiple instructions simultaneously when the instructions are not dependent on one another. Techniques such as speculative execution and register renaming are often used to uncover ILP in sequential code blocks.

As the granularity of parallelism is reduced, the inter-process communications requirement is typically increased. For this reason, not all levels of parallelism can be applied in a given system. For example, a distributed system is ill-suited to loop-level parallelism because the overhead of communication between processors can be much

larger than the time it would take a single processor to execute the loop itself. Similarly, ILP in a sequential program cannot be practically exposed by an SMP.

Traditional parallelization relies on a hierarchical system to exploit multiple levels of parallelism. As an example, consider a cluster of SMP nodes working on a 3D rendering job for a science-fiction film. Program-level parallelism can be used to work on multiple special-effects sequences simultaneously on different machines. Each machine may employ process-level parallelism to work on different frames of the same sequence in parallel. Thread-level parallelism might allow different regions of a single frame to be rendered in parallel. Within the threads, each processor exposes ILP automatically.

A CMP architecture allows parallelization at much smaller granularities than in traditional multiprocessors. With interprocessor communication on the order of a few clock cycles, thread-level parallelism with small path lengths and loop-level parallelism become practical. Uncovering such fine-grained parallelism requires new hardware and software mechanisms.

5.2 CMP-based Parallelization Studies

Several different approaches have been proposed to support fine-grained parallelism on a CMP. In this section, the required architectural resources for such support are examined under three different schemes: Multiscalar, Hydra, and Atlas. For each scheme, a brief description is provided along with a comparison of the overlap between the hardware required to support both MPE and fine-grained parallelism.

5.2.1 Multiscalar Chip-Multiprocessors

One approach to exploiting fine-grained parallelism is the multiscalar architecture [SOH95] from the University of Wisconsin. Though originally proposed for a multithreaded processor, the multiscalar approach has also been applied to a CMP [KRI98]. Multiscalar architectures rely on compiler support to identify parallel tasks. An otherwise sequential program is divided into a number of small threads that can be overlapped in execution to varying degrees.

Tasks may consist of portions of straight-line code, subroutine calls, or loop iterations. Task boundaries are selected such that each task begins and ends with a branch instruction. Each task is parsed for data that is required by successive tasks. Special instructions or tags on existing instructions are added to inform the underlying hardware when the last modification to this required data is made. When the special instruction is encountered, the data is passed to the successive task(s).

Tasks are identified through a Control Flow Graph (CFG) which is attached at compile-time to the binary executable and defines the task interaction of the program. A special hardware mechanism called a sequencer parses the CFG and determines which task(s) to start simultaneously. One drawback to multiscalar is that the CFG may need to be modified when the underlying hardware is changed to allow more or less degrees of parallelism.

In several respects, MPE behaves as a special case of multiscalar. In MPE, the confidence predictor identifies tasks bounded by low-confidence branch instructions. Whereas multiscalar applies compiler support to identify only those register dependencies that need to be communicated from one task to the next, MPE communicates all dependencies from the primary processor to the alternates.

The hardware requirements for multiscalar have been found to be very similar to that of MPE. In [KRI98], register values are communicated on a shared bus. It was found that a bus width of between one and three register-sized words and a latency of less than three clock cycles is necessary. In Section 4.4 of this dissertation, it was demonstrated that MPE requires between two and four register-sized words of interprocessor bandwidth and performs best with low communication latencies.

The multiscalar research efforts conclude that hardware support for parallelization of code at small granularities must include a memory disambiguation mechanism. Such hardware is used to detect memory value dependencies between threads that arise through the use of pointers that are otherwise impossible to detect at compile-time. The Address Resolution Buffer (ARB) [FRA96] and the Memory Disambiguation Table (MDT) [KRI98] represent two implementations of a memory disambiguation mechanism. Both techniques use a lookup table for memory load and store operations to determine if the operation will conflict with an outstanding load or store to the same memory address. To reduce the impact on resources and performance, the disambiguation may be applied in cache-line granularities and can make use of "safe" bits to indicate accesses that are known to be independent and can therefore avoid the table lookup.

In MPE, memory disambiguation is handled through a combination of cache coherency and explicit communication of all store operations. If a more ambitious memory disambiguation scheme such as the ARB or MDT is implemented to support multiscalar on a CMP, MPE can eliminate the need for explicit communication of store operations, reducing the bandwidth requirements of the interconnect.

5.2.2 Hydra Chip-Multiprocessor

Another example of exploiting fine-grained parallelism on a CMP can be found in the Hydra project [OPL97] at Stanford University. Hydra focuses on the architecture of a 4-processor CMP and the software support necessary to uncover loop-level parallelism in a similar manner as the multiscalar technique. With compiler support, separate iterations of certain loops are identified as being parallelizable and are executed simultaneously on different processors of the CMP.

Loop iterations need not be completely independent for Hydra to parallelize them. Special synchronization instructions are inserted to communicate loop-carried dependencies from one iteration to the next. Likewise, instructions are inserted to await these dependencies from threads working on previous iterations. Memory dependencies are avoided by locking critical regions that write to memory locations which cannot be resolved at compile time and coherency is kept by using write-through L1 caches.

This approach to data value synchronization is ill-suited to MPE due to the fact that the L1 caches are not updated with the latest value. Instead, the store-value synchronization applied for MPE might benefit the Hydra system by providing updated, coherent L1 caches for all processors.

Unlike the shared bus topology in the multiscalar study, [HAM97b] makes use of a unidirectional, ring-based interconnect for data value communication. This requirement results from the mapping of loop iterations to processors: data values need only be communicated from earlier iterations to later ones. In Section 4.5, it was shown that a ring-based interconnect is also well suited to MPE, provided that the links are bi-directional.

Interestingly, the Hydra performs best on floating-point benchmarks due to the regularity of branches and frequent use of loops inherent in such computationally-intensive applications. The speedup measured on integer benchmarks was very modest. Conversely, MPE is best suited to integer benchmarks due to unpredictable branch behavior. Therefore, support for both MPE and Hydra-style parallelism would be complimentary to improve performance on both types of applications.

5.2.3 Atlas Chip-Multiprocessor

The Atlas CMP [COD01] developed at the Georgia Institute of Technology combines fine-grained parallelism with data value speculation. Unlike the multiscalar and Hydra approaches where threads are identified on branch boundaries, Atlas divides sequential programs into tasks that begin with a load instruction and end with a store. The tasks are executed are executed simultaneously on different processors of the CMP.

Rather than requiring special instruction tags or synchronization instructions to handle intertask dependencies, Atlas makes use of a dependency predictor and a value predictor. The value predictor is used to guess the value of register input dependencies. A correct prediction allows the execution of a later task to fully overlap that of a prior task. An incorrect prediction results in the task being squashed and restarted with the correct input values. The dependency predictor maintains a history of each address to which a store is performed to determine if the stored value is loaded by a dependent task. Addresses that frequently are input dependencies to later tasks become subject to value prediction. The L1 data cache is also speculative in nature and detects mispredicted values through cache-coherence interaction with the other L1 caches.

Value prediction may be able to improve the performance of MPE. Rather than waiting on register values to be communicated from the primary processor, alternate

processors could speculatively execute instructions based on predicted values for the register input dependencies.

5.3 Summary

Supporting fine-grained parallelism shows much potential in a CMP environment. However, both software and hardware resources are necessary to enable such support. MPE, on the other hand, requires no changes to software but has less potential performance benefit.

In this chapter, it has been shown that three proposed methods for exploiting fine-grained parallelism on a CMP have similar architectural requirements as MPE. Therefore, a CMP that can support both MPE and fine-grained parallelism can be constructed with little additional hardware support. In some cases, the architectural requirements for fine-grained parallelism may enhance the performance of MPE.

Furthermore, prior studies into fine-grain parallelism demonstrate the best performance on floating-point applications for which MPE is ill-suited and are less effective on integer applications for which MPE shines. It seems likely that supporting both approaches will provide the highest performance on workloads that are a mixture of both types of applications.

CHAPTER 6
CONCLUSIONS

Designing future microprocessor is becoming increasingly difficult due to two factors: physical limitations and design complexity. Physical limitations arise from on-chip wiring delays and necessitate an emphasis on localized signaling. Design complexity results in exponentially increased complexity in the simulation and validation of designs. Chip-multiprocessors address both issues by integrating multiple processors on a single chip.

In the first part of this dissertation, the regular structure of a CMP was exploited to enable parallel simulation of CMP architectures. Because the amount of interprocessor communication in a CMP is kept to a minimum to avoid long, cross-chip signaling delays, the communication between nodes in the simulation platform is also limited. This effect enables use of a clustered system as the platform for the parallel simulation. Each processor of the CMP is simulated in a separate thread with the threads mapped to different physical processors in the cluster.

MPI was employed as the communication mechanism between threads, enabling the study of a wide variety of parallelization approaches and cluster interconnects. Through a series of microbenchmarks, the optimal combination was found prior to implementing a full CMP simulator. Due to the increased amount of aggregate cache in the compute nodes of a clustered system, the simulator performs better than the microbenchmark predictions for large systems, achieving a speedup of 12 on a 17-processor system.

In the second part of this dissertation, a simulative study of multiple-path execution explored the potential to achieve increased performance for unmodified, sequential applications running on a CMP. The MPE technique reduces the branch misprediction rate by simultaneously executing both paths of a conditional branch on different processors in the CMP. This technique is matched to the limited interprocessor bandwidth and realistic communication latencies between CMP processors.

The MPE study illustrates that a modest amount of interprocessor bandwidth is sufficient to yield average IPC speedups of almost 13% on SPECint95 with up to 34% speedup on applications with poor branch prediction accuracy. This level of speedup exceeds that obtained through conventional techniques such as issue-width scaling while permitting a faster clock frequency. It was also shown that MPE performs well with on-chip interconnect topologies that allow for relatively low-latency communication such as meshes and rings.

Finally, a comparison of the architectural support required for MPE and that of other studies of fine-grained parallelism on a CMP was found to be similar. Supporting both MPE and one or more other types of parallelism is projected to provide complementary performance increases for a wide variety of application types.

The research presented in this dissertation makes contributions to several areas of computer engineering. First, the parallel simulation techniques can be used to reduce the simulation time of complex multiprocessor systems such as CMPs. The microbenchmark-based evaluation technique can be adapted to a wide variety of parallel programs to narrow the design space in other complex applications. The MPE scheme presented in this dissertation will enable future CMP-based architectures to improve the

performance of unmodified, difficult-to-parallelize code. The analysis of MPE on a CMP can be used to guide the design requirements of such systems.

This dissertation has studied techniques for evaluating and designing processors in the relatively young yet promising area of chip-multiprocessor architectures. Novel architectural features coupled with a high-performance simulation environment will allow computer architects to continue the breakneck trend in computer performance scaling which has been a trademark of the electrical and computer engineering field over the past several decades.

LIST OF REFERENCES

[AHU98] P. Ahuja, K. Skadron, M. Martonosi, and D. Clark. Multipath execution: opportunities and limits. In *Proceedings of the 1998 International Conference on Supercomputing*, pp. 101-108, June 1998.

[AND95] T. Anderson, D. Culler, and D. Patterson. A case for NOW. *IEEE Micro*, 15(1), pp. 54-64, Feb. 1995.

[AUG98] D. August, D. Connors, S. Mahlke, J. Sias, K. Crozier, B. Cheng, P. Eaton, Q. Olaniran, and W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture,* pp. 227-237, July 1998.

[BOD95] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1), pp. 26-36, Jan. 1995.

[BUR97] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. Technical Report TR-1342, University of Wisconsin-Madison Computer Sciences Department, June 1997.

[CHA78] K. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5(5), pp. 440-452, 1979.

[COD99] L. Codrescu, M. Deb-Pant, T. Taha, J. Eble, S. Wills, J. Meindl. Exploring microprocessor architectures for gigascale integration. In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, pp. 242-255, 1999.

[COD01] L. Codrescu, D. Wills, and J. Meindl. Architecture of the Atlas chip-multiprocessor: dynamically parallelizing irregular applications. *IEEE Transactions on Computers*, 50(1), pp. 67-82, Jan. 2001.

[CUL99] D. Culler and J. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco: Morgan Kaufmann, 1999.

[DEV90] R. DeVries. Reducing null messages in Misra's distributed discrete event simulation method. *IEEE Transactions on Software Engineering*, 16(1), pp. 82-91, Jan. 1990.

[DUR99] M. Durbhakula, V. Pai, and S. Adve. Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pp. 23-32, Jan. 1999.

[EGG97] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, and D. Tullsen. Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro*, 17(5), pp. 12-19, Sept./Oct. 1997.

[FRA96] M. Franklin and G. Sohi. ARB: A hardware mechanism for dynamic memory disambiguation. *IEEE Transactions on Computers*, 45(5), pp. 51-67, May 1996.

[FUJ00] R. Fujimoto. *Parallel and Distributed Simulation Systems*. New York: John Wiley & Sons, Inc., 2000.

[GEO96] A. George and S. Cook. Distributed simulation of parallel DSP architectures on workstation clusters. *Simulation*, 67(2), pp. 94-105, Aug. 1996.

[GEO98] A. George, R. Fogarty, J. Markwell, and M. Miars. An Integrated Simulation Environment for parallel and distributed system prototyping. *Simulation*, 75(5), pp. 283-294, May 1999.

[HAM97a] L. Hammond, B. Nayfe, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9), pp. 79-85, Sept. 1997.

[HAM97b] L. Hammond and K. Olukotun. Considerations in the design of Hydra: A multiprocessor-on-a-chip microarchitecture. Technical report, Computer Systems Laboratory, Stanford University, 1997.

[HEN96] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. San Francisco: Morgan Kaufmann, 1996.

[HEN00] J. Henning. SPEC CPU2000: measuring CPU performance in the new millennium. *IEEE Computer*, 33(7), pp. 28-35, July 2000.

[JAC96] E. Jacobsen, E. Rotenberg, and J. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 142-152, Dec. 1996.

[JOH96] R. Johnson and M. Schlansker. Analysis techniques for predicated code. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 100-113, Dec. 1996.

[JOU97] S. Jourdan, J. Stark, T.-H. Hsing, and Y. Patt. Recovery requirements of branch prediction storage structures in the presence of mispredicted-path execution. *International Journal on Parallel Programming*, 25(5), pp. 363-383, Oct. 1997.

[KEC98] S. Keckler. Fast thread communication and synchronization mechanisms for a scalable single chip multiprocessor. Ph.D. Thesis, Massachusetts Institute of Technology, June 1998.

[KES98] R. Kessler, E. McLellan, and D. Webb. The Alpha 21264 microprocessor architecture. In *Proceedings of the 1998 International Conference on Computer Design: VLSI in Computers and Processors*, pp. 250-259, June 1998.

[KLA99] A. Klauser and D. Grunwald. Instruction fetch mechanisms for multipath execution processors. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp 38-47, Nov. 1999.

[KLA98] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the PolyPath architecture. In *Proceedings of the 25th International Symposium on Computer Architecture,* pp. 250-259, July 1998.

[KOZ97] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. *IEEE Computer*, 30(9), pp. 75-78, Sept. 1997.

[KRI98] V. Krishnan and J. Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip multiprocessor. In *Proceedings of the ACM 1998 International Conference on Supercomputing*, pp. 85-92, June 1998.

[LIP97] M. Lipasti, J. Shen. Superspeculative microarchitecture for beyond A.D. 2000. *IEEE Computer*, 30(9), pp. 59-66, Sept. 1997.

[MAT97] D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9), pp. 37-39, Sept. 1997.

[MCF93] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[MPI94] MPI Forum. *MPI: A Message-Passing Interface Standard.* WWW site: www.mpi-forum.org, June 22, 1994.

[MUK97] S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. Hill, J. Larus, and D. Wood. Wisconsin Wind Tunnel II: A fast and portable parallel architecture simulator. In *Workshop on Performance Analysis and Its Impact on Design*, June 1997.

[OLU96] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, K. Chung. The case for a single-chip multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2-11, Oct. 1996.

[OPL97] J. Oplinger, D. Heine, S. Liao, B. Nayfeh, M. Lam, and K. Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Stanford University Computer Systems Lab, Feb. 1997.

[PAI97] V. Pai, P. Ranganathan, and S. Adve. *RSIM Reference Manual version 1.0*, Technical Report 9705, Department of Electrical and Computer Engineering, Rice University, Aug. 1997.

[PAT97] Y. Patt, S. Patel, M. Evers, D. Friendly, and J. Stark. One billion transistors, one uniprocessor, one chip. *IEEE Computer*, 30(9), pp. 51-57, Sept. 1997.

[PRI95] C. Price. *MIPS IV Instruction Set, Revision 3.1.* MIPS Technologies, Inc., Mountain View, CA, Jan. 1995.

[REI98] M. Reilly and J. Edmondson. Performance simulation of an Alpha microprocessor. *IEEE Computer*, 31(5), pp. 50-58, May 1998.

[ROT96] E. Rotenberg, S. Bennet, and J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 24-34, Dec. 1996.

[SAZ97] Y. Sazeides and J. Smith. The predictability of data values. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 248-258, Dec. 1997.

[SCA00] Scali Computer AS. Scali System Guide version 2.0, white paper. WWW site: www.scali.com, July 13, 2000.

[SCI93] *Scalable Coherent Interface: ANSI/IEEE Standard 1596-1992.* Piscataway, NJ: IEEE Service Center, 1993.

[SKA99] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark. Branch prediction, instruction-window size, and cache size: performance tradeoffs and simulation techniques. *IEEE Transactions on Computers*, 48(11), pp. 1260-1281, Nov. 1999.

[SOD97] A. Sodani and G. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 194-205, June 1997.

[SOH95] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 414-425, 1995.

[SPE96] The Standard Performance Evaluation Corporation. WWW site: www.specbench.org, Dec. 10, 1996.

[TOM67] R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1), pp. 25-33, Jan. 1967.

[UHT95] A. Uht and V. Sindagi. Disjoint eager execution: an optimal form of speculative execution. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 313-325, Dec. 1995.

[VEE94] J. Veenstra and R. Fowler. *MINT Tutorial and User Manual*, Technical Report 452, Department of Computer Science, University of Rochester, Aug. 1994.

[WAL93] D. Wall. Limits of instruction-level parallelism. Technical Report 93/9, Digital Western Research Laboratory, Nov. 1993.

[WAL98] S. Wallace, B. Calder, and D. Tullsen. Threaded multiple path execution. In *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 238-249, June 1998.

[WOO95] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 24-36, June 1995.

[YEA96] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2), pp. 28-40, Feb. 1996.

[YEH93] T. Yeh and Y. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 257-266, May 1993.

BIOGRAPHICAL SKETCH

Matthew Chidester was born in Leonardtown, Maryland, but soon moved to Longwood, Florida, at the age of 3 months where he spent his entire childhood. After graduating as the valedictorian of Lake Brantley High School in 1994, he attended the University of Florida where he was awarded a Bachelor of Science in Electrical Engineering under the BS/MS program in 1997. Matthew was honored with an Electric "E" Award and was recognized as an Outstanding Four-Year Scholar. He is also a member of the IEEE and the Eta Kappa Nu Electrical Engineering Honorary society.

Matthew entered the graduate program at the University of Florida in January of 1997 and joined the High-performance Computing and Simulation (HCS) Research Laboratory where he has been involved in many projects, including high-performance network modeling and testbed experiments, processor/memory interface modeling and simulation, clustered computing, and chip-multiprocessing. He served as team leader for the High-performance Clusters (HPC) team beginning in the spring of 1999.

Matthew's research was supported in part by a National Science Foundation graduate research fellowship. After receiving his Ph.D., Matthew will be taking a position as a Senior Component Design Engineer with Intel in Hillsboro, OR.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Alan D. George, Chairman
Associate Professor of Electrical and Computer Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Michael P. Frank
Assistant Professor of Computer and Information Sciences and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Mark E. Law
Professor of Electrical and Computer Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Stanley Y. Su
Professor of Computer and Information Sciences and Engineering and Electrical and Computer Engineering

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

August 2001

Pramod P. Khargonekar
Dean, College of Engineering

Winfred M. Phillips
Dean, Graduate School